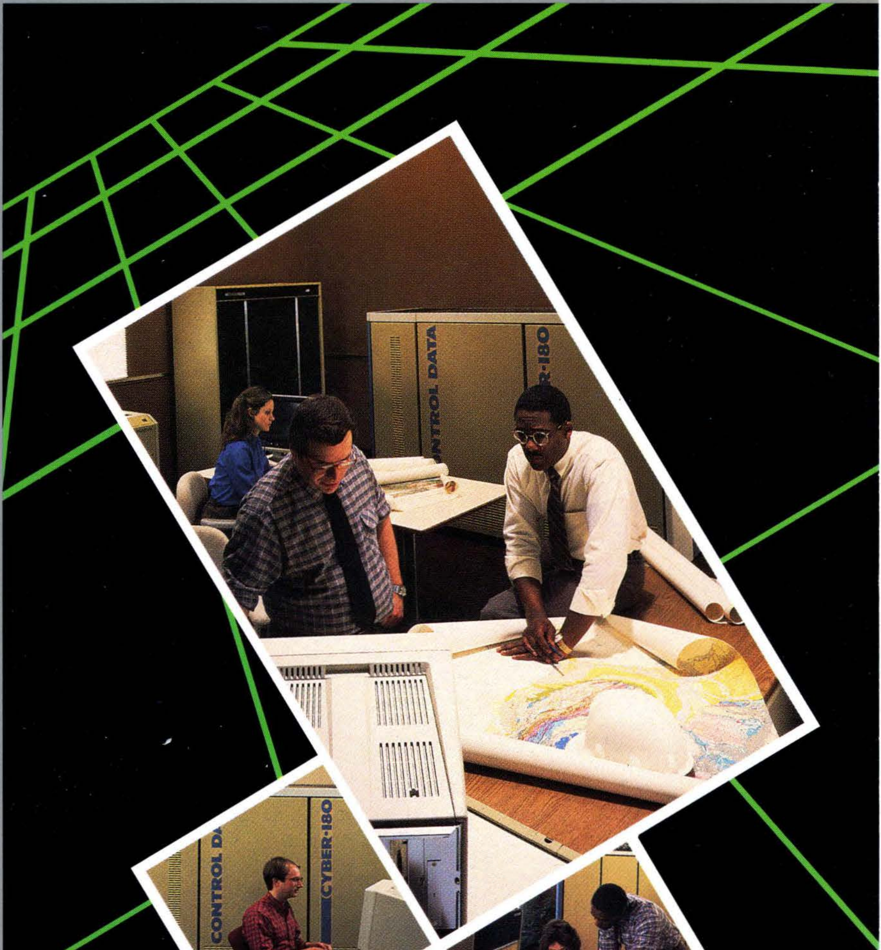
CYBIL for NOS/VE Language Definition

CONTROL DATA



Usage

60464113

Keyword Index

Statement Index

Function Index

(Continued on inside back cover

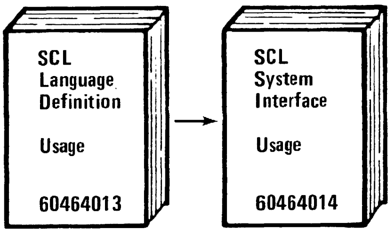
CYBIL for NOS/VE Language Definition

Usage

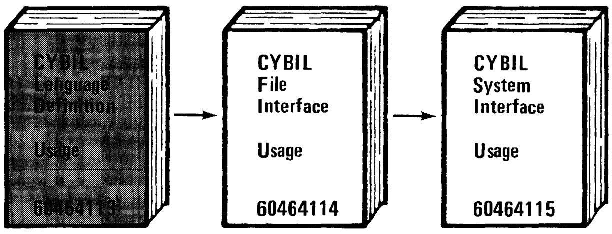
Publication Number 60464113

Related Manuals

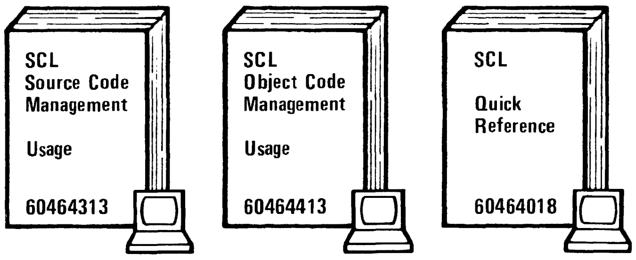
Background (Access as Needed):


SCL Language Definition Usage 60464013
SCL System Interface Usage 60464014

CYBIL Manual Set:


CYBIL Language Definition Usage 60464113
CYBIL File Interface Usage 60464114
CYBIL System Interface Usage 60464115

Additional References:


SCL Source Code Management Usage 60464313
SCL Object Code Management Usage 60464413
SCL Quick Reference 60464018

→ indicates the reading sequence.

🖥 indicates an online version of the manual is available.

Manual History

This manual is Revision B, printed in July 1984. It reflects NOS/VE Version 1.1.1. at PSR level 613. Minor technical corrections and editorial changes have been incorporated. New Debug commands and functions have been added.

Previous Revision	System Version	Date
A	1.0.2.	February 1984

Contents

About This Manual

This manual describes CYBIL, the implementation language of the CONTROL DATA® Network Operating System/Virtual Environment (NOS/VE).

Audience

This manual is written as a reference for CYBIL programmers. It assumes that you understand NOS/VE and System Command Language (SCL) concepts as presented in the SCL Language Definition manual and the SCL System Interface manual. You will also need to be familiar with the CYBIL File Interface manual in order to perform input to and output from a CYBIL program.

Organization

This manual is organized by topic, based on elements of the CYBIL language. The first chapter introduces the basic elements of the language and refers you to the chapter in which each is further described.

This manual is part of the CYBIL manual set. The CYBIL File Interface manual describes the facilities available to read and write files used by CYBIL. The CYBIL System Interface manual describes the CYBIL procedures you can call to use the special capabilities of NOS/VE.

Conventions

Within the formats for declarations, type specifications, and statements shown in this manual, uppercase letters represent reserved words; they must appear exactly as shown. Lowercase letters represent names and values that you supply.

Required parameters are shown in bold type. Optional parameters are shown in italics and are enclosed by braces, as in:

{ *PACKED* }

If the parameter is optional and can be repeated any number of times, it is also followed by several periods, as in:

{ *name* }...

For example, the notation {*digit*} means zero digits or one digit can appear; {*digits*}... means zero, one, or more digits can appear. Braces indicate that the enclosed parameters are used together. For example,

{*offset MOD base*}

is considered a single parameter. Except for the braces and periods indicating repetition, all other symbols shown in a format must be included.

Numbers are assumed to be decimal unless otherwise noted.

In examples that show interactive terminal sessions, user input is printed in blue. System output is printed in black.

New features, as well as changes, deletions, and additions to information in this manual, are indicated by vertical bars in the margins or by a dot near the page number if more than half the page is affected.

Additional Related Manuals

The related manuals listed on page 2 include the manuals you should be familiar with to this point, and which manuals you may want to read following this one. In addition, you may want to have a copy of the CDC® CYBER 170/180 Models 810, 815, 825, 830, 835, 845, 855, and 990 (Virtual State) Hardware Reference Manual, Volume II, publication number 60458890. You do not need the hardware manual to use the information in this manual, but it provides you with more detail about the hardware and, in particular, the hardware instructions used in certain CYBIL procedures described in this manual.

The Math Library manual, publication number 60486513, describes the mathematical routines available in the Math Library. These routines can be accessed by CYBIL programs.

The Diagnostic Messages for NOS/VE manual, publication number 60464613, documents diagnostic messages generated by NOS/VE.

Ordering Manuals

Control Data manuals are available through Control Data sales offices or through:

Control Data Corporation
Literature Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Submitting Comments

The last page of this manual is a comment sheet. Please use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report technical or typographical errors. If the comment sheet has already been used, you can mail your comments to:

Control Data Corporation
Publications and Graphics Division ARH219
4201 Lexington Avenue North
St. Paul, Minnesota 55112

Please indicate whether you would like a written response.

Introduction 1

This chapter introduces the basic elements of a CYBIL program and refers you to the chapter in which each is further described.

Introduction 1

A CYBIL program consists essentially of two kinds of elements: declarations and statements. Declarations describe the data to be used in the program. Statements describe the actions to be performed on the data.

Declarations and statements are made up of predefined reserved words and user-defined names and values. The way you form these elements is described in chapter 2, as is the general structure for forming a CYBIL program.

Data can be either constant or variable. You can use the constant value itself or give it a name using the constant declaration (CONST). Variables are named, initialized, and given certain characteristics with the variable declaration (VAR). One of the characteristics of a variable is its type, for example, integer or character. You can use CYBIL's predefined types or define your own types. To define a new type or redefine an existing type with a new name, you use the type declaration (TYPE). Once you have defined a type, CYBIL will treat it as a standard data type; you can specify your new type name as a valid type in a variable declaration and CYBIL will perform standard type checking on it. You can also declare where you want certain variables to reside by defining an area called a section, which can be a read-only section or a read/write section. This is done with the SECTION declaration. All of these data-related declarations are described in chapter 3.

Many standard types are available, including integers, floating-point numbers, characters, and boolean values, to name a few. In addition, you can use combinations of the standard types to define your own data types, for example, a record that contains several fields. The next few paragraphs summarize the types that are predefined by CYBIL. They are described in detail in chapter 4.

Among the basic types are scalar types, that is, those that have a specific order. Besides integer, character, and boolean values, you can declare an ordinal type in which you define the elements and their order. You can also specify a subrange of any of the scalar types by giving a lower and upper bound. Floating-point (real) numbers are also available. A cell, which represents the smallest addressable unit of memory, can be specified as a type. A pointer is a type that points to a variable, allowing you to access the variable by location rather than by name. These are the basic types: scalar, floating point, cell, and pointer. With these basic types you can construct the structured types: strings, arrays, records, and sets.

A string is a sequence of characters. You can reference a portion of a string (called a substring) or a single character within a string. An array is a structure that contains components all of the same type. The components of an array have a specific order and each one can be referenced individually. A record is a structure that contains a fixed number of fields, which may be of different types. Each field has a unique name within the record and can be referenced individually. You can also declare a variant record that has several possible variations (variants). The current value of a field common to all variants, or the latest assignment to a specific variant field determines which of the variants should be used for each execution. A set is a structure that contains elements of a single type. Yet unlike an array, elements in a set have no order and individual elements cannot be referenced. A set can be operated on only as a whole.

Storage types are structures to which variables can be added, referenced, and deleted under explicit program control using a set of storage management statements. The two storage types are sequences and heaps.

All of the types mentioned above are considered fixed types; that is, there is a definite size associated with each one when it is declared. If you want to delay specifying a size until execution time, you can declare it as an adaptable type. Then, sometime during execution, you assign a fixed size or value to the type. A string, array, record, sequence, or heap can be adaptable.

All of these types are described in chapter 4.

Statements define the actions to be performed on the data you've defined. The assignment statement changes the value of a variable. Structured statements contain and control the execution of a list of statements. The BEGIN statement unconditionally executes a statement list. The WHILE, FOR, and REPEAT statements control repetitive executions of a statement list.

Control statements control the flow of execution. The IF and CASE statements execute one of a set of statement lists based on the evaluation of a given expression or the value of a specific variable. CYCLE, EXIT, and RETURN statements stop execution of a statement list and transfer control to another place in the program.

Storage management statements allocate, access, and release variables in sequences (using the RESET and NEXT statements), heaps (using the RESET, ALLOCATE, and FREE statements), and the run-time stack (using the PUSH statement).

All of the preceding statements are described in detail in chapter 5, along with the operands and operators that can be used in expressions within statements and declarations.

Statements can appear within a program (as described in chapter 2), a function, or a procedure.

A function is a list of statements, optionally preceded by a list of declarations. It is known by a unique name and can be called by that name from elsewhere in the program. A function performs some calculation and returns a value that takes the place of the function reference. There are many standard functions defined in CYBIL and you can also create your own. Standard functions and rules for forming your own functions are described in chapter 6.

A procedure, like a function, is a list of statements, optionally preceded by a list of declarations. It also is known by a unique name and can be called by that name from elsewhere in the program. A procedure performs specific operations and may or may not return values to existing variables. You can use the standard procedures and also define your own. Chapter 7 describes the standard procedures and rules for forming your own procedures.

Chapter 8 describes the CYBIL command you use to call the CYBIL compiler, tell it which files to use for input and output, and specify what kind of listing you want. It also describes directives that are available at compilation time to specify listing options, run-time options, the layout of the source text and resulting object listing, and what specific portions of the source text to compile.

Chapter 9 describes the Debug utility, which aids you in debugging CYBIL programs at a source code level or machine code level, in either interactive or batch mode.

In summary, chapters 2 through 7 describe the elements within a CYBIL program. Chapter 8 describes the command and directives that control how the program is actually compiled. Chapter 9 describes debugging capabilities.

Procedures that perform input to and output from CYBIL programs are described in the CYBIL File Interface manual.

Program Structure 2

This chapter describes how to form the individual elements used within a program and how to structure the program itself.

Program Structure 2

This chapter describes how to form the individual elements used within a program and how to structure the program itself.

Elements Within a Program

Valid Characters

The characters that can be used within a program are those in the ASCII character set that have graphic representations (that is, can be printed). This character set is included in appendix B. It contains uppercase and lowercase letters. In names that you define, you can use uppercase and lowercase letters interchangeably. For example, the name LOOP_COUNT is equivalent to the name loop_count.

CYBIL-Defined Elements

CYBIL has predefined meanings for many words and symbols. You cannot redefine or use these words and symbols for other purposes.

A complete list of CYBIL reserved words is given in appendix C. In the formats for declarations, type specifications, and statements shown in this manual, reserved words are shown in uppercase letters.

The following list includes the reserved symbols and a brief description of the purpose of each. They are discussed in more detail throughout this manual.

Symbol	Purpose
+, -, *, /, =, <, <=, >, >=, < >, :=, (,)	These symbols are primarily operators used in expressions. They are discussed in chapter 5.
;	The semicolon separates individual declarations and statements.
:	The colon is used in declarations as described in chapter 3.
,	The comma separates repeated parameters or other elements.
.	A single period indicates a reference to a field within a record as described in chapter 4.

(Continued)

(Continued)

Symbol	Purpose
..	Two consecutive periods indicate a subrange as described in chapter 4.
^	The circumflex indicates a pointer reference as described in chapter 4.
' '	Apostrophes delimit strings.
[ ]	Brackets enclose array subscripts, indefinite value constructors, and set value constructors as described in chapter 4.
{ }	Braces delimit comments. (Within the formats shown in this manual, they are also used to enclose optional parameters.)
? or ??	A single question mark or a pair of consecutive question marks indicate compile-time statements and directives as described in chapter 8.

User-Defined Elements

Names

You define the names for elements, such as constants, variables, types, procedures, and so on, that you use within a program. A name:

- Can be from 1 through 31 characters in length.
- Can consist of letters, digits, and the special characters # (number sign), @ (commercial at sign), _ (underline), and $ (dollar sign).†
- Must begin with a letter. (There is an exception to this rule for system-defined functions and procedures that begin with the # or $ character.)
- Cannot contain spaces.

† NOS/VE often uses $ in its predefined names. To keep from matching a system reserved name, avoid using $ in the names you define.

In the formats included in this manual, names that you supply are shown in lowercase letters. Within a program, however, there is no distinction between uppercase and lowercase letters. The name my_file is identical to the name My_File.

There is considerable flexibility in forming names, so you should make them as descriptive as possible to promote readability and maintainability of the program. For example, LAST_FILE_ACCESSED is more obvious than LASTFIL.

Examples:

Valid Names	Invalid Names
SUM	ARRAY
REGISTER#3	FILES&POSITIONS
POINTER_TABLE	2ND

The valid names need no explanation. Among the invalid names, ARRAY cannot be used because it is a reserved word; FILES&POSITIONS contains an invalid character (the ampersand); and 2ND does not begin with a letter.

Constants

A constant is a fixed value. It is known at compilation time and does not change throughout the execution of a program. It can be an integer, character, boolean, ordinal, floating-point number, pointer, or string.

Integer constants can be binary, octal, decimal, or hexadecimal. The base is specified by enclosing the radix in parentheses following the integer, as follows:

integer (radix)

Examples are 1011(2) and 19A(16). If the radix is omitted, the integer is assumed to be decimal. Integer constants must start with a digit; therefore, zero must precede any hexadecimal constant that would otherwise begin with a letter, for example, 0FF(16). Negative integer constants must be preceded by a minus sign. Positive integer constants can be preceded by a plus sign but need not be.

Integer constants range in value from $-(2^{63}-1)$ through $2^{63}-1$.

A character constant can be any single character in the ASCII character set. The character is enclosed in apostrophes in the following form:

'character'

Examples are 'A' and '?'. The apostrophe character itself is specified by a pair of apostrophes.

A boolean constant can be either FALSE or TRUE, each having its usual meaning.

An ordinal constant is an element of an ordinal type that you have defined. For further information, refer to Ordinal under Scalar Types in chapter 4.

Floating-point (real) constants can be written in either decimal notation or scientific notation. A real number written in decimal notation contains a decimal point and at least one digit on each side, for example, 5.123 or -72.18. If the number is positive, the sign is optional; if negative, the sign is required.

A real number written in scientific notation is represented by a number (the coefficient), which is multiplied by a power of 10 (the exponent) in the form:

```
coefficientEexponent
```

The prefix E is read as "times 10 to the power of "; for example,

```
5.1E6
```

is 5.1 times 10 to the power of 6, or 5,100,000. The decimal point in the coefficient is optional. A decimal point cannot appear in the exponent; it must be a whole number. If the coefficient or exponent is positive, the sign is optional; if negative, the sign is required.

The pointer constant is NIL. It indicates an unassigned pointer. NIL can be assigned to a pointer of any type.

String constants consist of one or more characters enclosed in apostrophes in the following form:

```
'string'
```

An example is 'USER1234', a string of eight characters. An apostrophe in a string constant is specified by a pair of apostrophes, for example, 'DON''T'.

String constants can be concatenated by using the reserved word CAT, as in:

```
'characters_1' CAT 'characters_2'
```

The result is the string 'characters_1characters_2'. The CAT operation cannot be used with string variables.

A string constant can be empty, that is, a null string; for example,

```
str := '';
```

assigns a null string to the string constant STR.

You cannot reference parts (substrings) of string constants.

Constant Expressions

Expressions are combinations of operands and operators that are evaluated to find scalar or string type values. In a constant expression, the operands must be constants, names of constants (that you declare using the CONST declaration described in chapter 3), or other constant expressions within parentheses. Computation is done at compile time and the resulting value used in the same way a constant is used.

The general rules for forming and evaluating expressions are described under Expressions in chapter 5. These rules apply to constant expressions with the following exceptions:

- Constant expressions must be simple expressions; terms involving relational operators must be delimited with parentheses.
- The only functions allowed as factors in constant expressions are the $INTEGER, $CHAR, SUCC, and PRED functions with constant expressions as arguments.
- Substring references are not allowed.

Syntax

The exact syntax of the language is shown in the formats of individual declarations and statements described in the remainder of this manual. The following paragraphs discuss general syntax rules.

Spaces

Spaces can be used freely in programs with the following exceptions:

- Names and reserved words cannot contain embedded spaces. Normally, constants cannot contain spaces either, but a character constant or string constant can.
- A name, reserved word, or constant cannot be split over two lines; it must appear completely on one line.
- Names, reserved words, and constants must be separated from each other by at least one space, or one of the other delimiters such as a parenthesis or comma.

For further information, refer to Spacing later in this chapter.

Comments

Comments can be used in a program anywhere that spaces can be used (except in string constants). They are printed in the source listing but otherwise are ignored by the compiler.

A comment is enclosed in left and right braces: { }. It can contain any character except the right brace (}). To extend a comment over several lines, repeat the left brace ({) at the beginning of each line. If the right brace is omitted at the end of the comment, the compiler ends it automatically at the end of the line.

Example:

```
{this comment
{appears on
{several lines.}
```

Within this manual, the formats for declarations, type specifications, and statements use braces to indicate an optional parameter.

Punctuation

A semicolon separates individual declarations and statements. It must be included at the end of almost every declaration and statement. The single exception is MODEND which can, but need not, end with a semicolon if it is the last occurrence of MODEND in a compilation. Punctuation for specific declarations and statements is shown in the formats in the following chapters.

Two consecutive semicolons indicate an empty statement, which the compiler ignores. Spacing between the semicolons in this case is unimportant.

Spacing

Declarations and statements can start in any column. In this manual, indentations are used in examples to improve readability. It is recommended that similar conventions be used in your programs to aid in debugging and documentation for yourself and other users.

The LEFT and RIGHT directives, described in chapter 8, can be used at compilation time to specify the left and right margins of the source text. All source text outside of those margins is then ignored. A warning diagnostic is issued for every line that exceeds the specified right margin.

A name, reserved word, or constant cannot be split over two lines; each must appear completely on one line.

Structure of a Program

Module Structure

The basic unit that can be compiled is a module and, optionally, compile-time statements and directives. A module can, but need not, contain a program. The general structure of a module is:

```
MODULE module_name;
   declarations
   PROGRAM program_name;
      declarations
      statements
   PROCEND program_name;
MODEND module_name;
```

Declarations can be constant, type, variable, section, function, and procedure declarations. A module can contain any number and combination of declarations, but it can contain at most one program. The program contains the code (that is, the statements) that are actually executed. The required module and program declarations are described later in this chapter.

The structure within a module determines the scope of the elements you declare within it.

Scope

The scope of an element you declare, such as a variable, function, or procedure, is the area of code where you can refer to the element and it will be recognized. Scope is determined by the way the program and procedures are positioned in a module and where the elements are declared.

In terms of scope, the programs, procedures, and functions are often referred to as blocks (that is, blocks of code). Generally, if an element is declared within a block, its scope is just that block. Outside the block, the element is unknown and references to it are not valid. A variable declared within a block is said to be local to the block and is called a local variable.

An element declared at the module level (that is, one that is not declared within a program, procedure, or function) has a scope of the entire module. It can be referred to anywhere within the module. A variable declared at the module level is said to be global and is called a global variable.

A block can contain one or more subordinate blocks. A variable declared in an outer block can always be referenced in a subordinate block. However, if a subordinate block declares an element of the same name, the new declaration applies while inside that block. Figure 2-1 illustrates these rules.

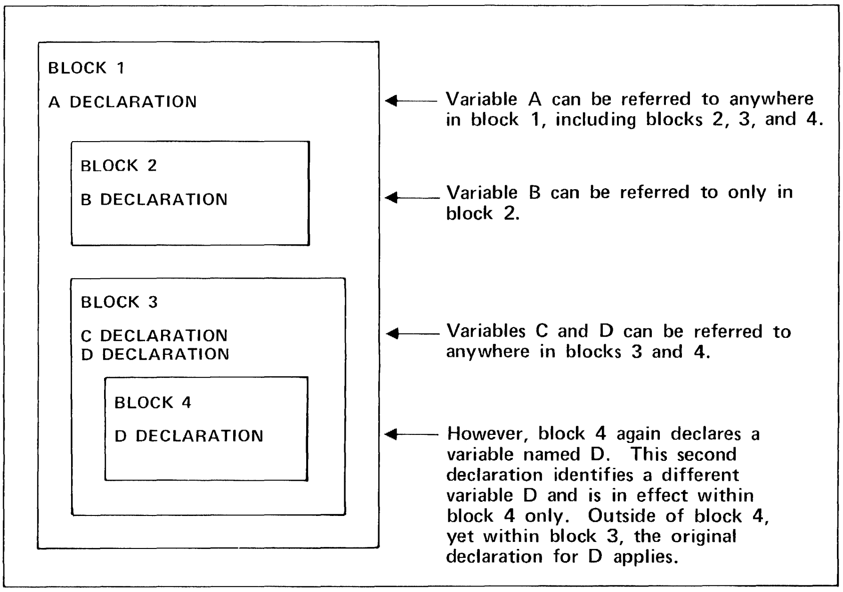

BLOCK 1
A DECLARATION
BLOCK 2
B DECLARATION
BLOCK 3
C DECLARATION
D DECLARATION
BLOCK 4
D DECLARATION
Variable A can be referred to anywhere in block 1, including blocks 2, 3, and 4.
Variable B can be referred to only in block 2.
Variables C and D can be referred to anywhere in blocks 3 and 4.
However, block 4 again declares a variable named D. This second declaration identifies a different variable D and is in effect within block 4 only. Outside of block 4, yet within block 3, the original declaration for D applies.

Figure 2-1. Scope of Variables Within a Block Structure

Storage space is allocated for a variable when the block in which it is declared is entered. Space is released when an exit is made from the block. Because space is allocated and released automatically, these variables are called automatic variables. You can specify that storage for a variable remains throughout execution by including the STATIC attribute when you declare the variable. A variable declared in this way is called a static variable. A global variable is always static. Because it is declared at the outermost level of a module (consider the module to be a block), storage for a global variable is allocated throughout execution of the module (or block). For further information on automatic and static variables, refer to Variable Declaration in chapter 3.

The one exception to the preceding rules is an element declared with the XDCL (externally declared) attribute. This attribute means the element is declared in one module but can be referred to in another. In this case, the loader handles the links between modules. For further information on the XDCL attribute, refer to chapter 3.

Module Declaration

The module declaration marks the beginning of a module. MODEND marks the end of a module. A module can contain at most one program and any combination of type, constant, variable, section, function, and procedure declarations. If two or more modules are compiled and linked together for execution, there can be only one program declaration in all the linked modules.

The format of the module declaration is:

MODULE name;†

name
The name of the module.

The format of MODEND is:

MODEND { *name* };

name
The name of the module. This parameter is optional. If used, the name must be the same as that specified in the module declaration.

When compiling more than one module, a semicolon is required after each occurrence of MODEND except the last one. There it is not required but is recommended.

Examples:

The following example shows a module named ONE that contains various declarations and a program named MAIN. The module name and semicolon could be omitted following MODEND, but it is recommended that they both be included.

```
MODULE one;
  declarations
  PROGRAM main;
    declarations
    statements
  PROCEND main;
MODEND one;
```

† Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a module declaration. If included in a CYBIL program run on NOS/VE, this parameter is ignored.

The following example shows a compilation consisting of three modules named ONE, TWO, and THREE. All three modules can be compiled and the resulting object modules linked together to form a single object module that can then be executed. For readability, the module names are included in all occurrences of MODEND. The semicolon could be left off the last occurrence of MODEND, but it is a good practice to include it.

```
MODULE one;
  declarations/statements
MODEND one;
MODULE two;
  declarations/statements
MODEND two;
MODULE three;
  declarations/statements
MODEND three;
```

Program Declaration

The program declaration marks the beginning of a program. The end of a program is marked by a PROCEND statement. A program can contain any combination of type, constant, variable, section, function, and procedure declarations, and any statements. If two or more modules are compiled and linked together for execution, there can be only one program declaration in the linked modules.

The format of the program declaration is:

PROGRAM name {*(formal_parameters)*};†

name

The name of the program.

formal_parameters

One or more optional parameters included if the program is to be called by the operating system. They can be in the form

VAR name {,*name*}... **: type**
{,*name* {,*name*}... : *type*}...

and/or

name {,*name*}... **: type**
{,*name* {,*name*}... : *type*}...

where **name** is the name of the parameter and **type** is the type of the parameter, that is, a predefined type (described in chapter 4) or a user-defined type (described in chapter 3).

The first form is called a reference parameter; its value can be changed during execution of the program. The second form is called a value parameter; its value cannot be changed by the program. Both kinds of parameters can appear in the formal parameter list; if so, they are separated by semicolons (for example, I: INTEGER; VAR A: CHAR). Reference and value parameters are discussed in more detail later in this chapter.

† Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a program declaration. If included in a CYBIL program run on NOS/VE, this parameter is ignored.

The optional parameter list is included if a CYBIL program is to be called by the operating system. It allows the system to pass values (for example, a string that represents a command) to a CYBIL program. When the system calls a program, it includes parameters called actual parameters in the call. The values of those actual parameters replace the formal parameters in the parameter list one-for-one based on position; that is, the first actual parameter replaces the first formal parameter, and so on. Wherever the formal parameters appear in statements within the program, the values of the corresponding actual parameters are substituted. For every formal parameter in the program declaration, there must be a corresponding actual parameter.

When a reference parameter is used, the formal parameter represents the corresponding actual parameter throughout execution of the program. Thus, an assignment to a formal parameter changes the variable that was passed as the corresponding actual parameter. An actual parameter that corresponds to a formal reference parameter must be addressable. A formal reference parameter can be of any type.

When a value parameter is used, the formal parameter takes on the value of the corresponding actual parameter. However, the program cannot change a value parameter by assigning a value to it or specifying it as an actual reference parameter to a procedure or function. A formal value parameter can be of any type except a heap, or an array or record that contains a heap.

The format of PROCEND is:

PROCEND { *name* };

name

The name of the program. This parameter is optional. If used, the name must be the same as that specified in the program declaration.

Example:

The following example shows a program named MAIN that contains various declarations, including a procedure named SUB_1.

```
PROGRAM main;
  declarations
  PROCEDURE sub_1;
    declarations
    statements
  PROCEND sub_1;
  statements
PROCEND main;
```

Constant, Variable, Type, and Section Declarations 3

This chapter describes how you declare constant and variable data types and new data types. It also describes how you specify a particular section in which to group data.

Constant, Variable, Type, and Section Declarations 3

This chapter describes the constant declaration, which defines a name for a value that never changes; the variable declaration, which defines a name for a value that can change; and the type declaration, which defines a new type of data and gives a name to that type. In addition, it also describes the section declaration, which groups variables that share common access characteristics.

Constant Declaration

A constant, as described in chapter 2, is a fixed value that is known at compile time and doesn't change during execution. A constant declaration allows you to associate a name with a value and use that name instead of the actual constant value. This provides greater readability because the name can be descriptive of the constant. Constant declarations also provide greater maintainability because the constant value need only be changed in one place, the constant declaration, not every place it is used in the code.

The format of the constant declaration is:

CONST name = value {,*name* = *value*}...;

name

The name associated with the constant value.

value

The constant value. It can be an integer, character, boolean, ordinal, floating-point, pointer, string, or constant expression. Rules for forming these values are given under Constants and under Constant Expressions in chapter 2.

You can write several constant declarations, each declaring a single constant, or a single declaration declaring several constants where each **name = value** combination is separated by a comma.

Type is not specified in a constant declaration. The type of the constant is the same as the type of the value assigned to it.

If used, an expression is evaluated during compilation. The expression itself can contain other constants.

Examples:

Rather than repeat the value of pi throughout a program, you can use a constant declaration to assign a descriptive name (in this case, PI) to the value and use that name in subsequent expressions and operations. The constant declaration is:

```
CONST pi = 3.1415927;
```

The following example shows a constant declaration containing several different types.

```
CONST
  first = 1,
  last = 80,
  hex = 0A8(16),
  bit_pattern = 10110101(2),
  fp_number = 1.2E3,
  stop_character = '.',
  continue = TRUE,
  message = 'end of line',
  last_pointer = NIL,
  length = last - first,
  result = (1 * 2) DIV 3;
```

Each constant has the same type as the value assigned to it. For example, FIRST and LAST are integer types, as is LENGTH, which is the result of an expression containing integers. Notice that the value of HEX begins with a 0 (zero) because integers must begin with a digit.

Variable Declaration

A variable is an element within a program whose value can change during execution. The name of the variable stays the same; it is only the value contained in the variable that changes. To use a variable, you must declare it.

The format for a variable declaration is:

VAR name {*,name*}... **:** {*[attributes]*} **type** {*:= initial_value*}
{*,name* {*,name*}... *:* {*[attributes]*} *type* {*:= initial_value*}}...**;**†

name

The name of the variable. Specifying more than one name indicates that all of the named variables will have the characteristics that follow (attributes, type, and initial_value).

attributes

One or more of the following attributes. If more than one are specified, they are separated by commas.

READ

Access attribute specifying that the variable is a read-only variable; the compiler checks to ensure that the value of the variable is not changed. If READ is specified, an initial value is required.

XDCL

Scope attribute specifying that the variable is declared in this module but can be referenced from another module.

XREF

Scope attribute specifying that the variable is declared in another module but can be referenced from this module.

#GATE††

Scope attribute that allows the variable to be accessed by a procedure at a higher ring level. If #GATE is specified, the XDCL attribute is required also.

† Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a variable declaration. If included in a CYBIL program run on NOS/VE, this parameter is ignored.

†† This attribute is not supported on variations of CYBIL available on other operating systems.

STATIC

Storage attribute specifying that storage space for the variable is allocated at load time and remains when control exits from the block. Static storage is assumed when any attributes are specified.

section_name

Storage attribute specifying the name of the section in which the variable resides. A variable in a read-only section is protected by hardware, as opposed to software. The section name and its read/write attributes must be declared using the section declaration (discussed later in this chapter).

Attributes are described in more detail later in this chapter.

The attributes parameter is optional. If omitted, CYBIL assumes the variable can be read and written; can be referenced only within the block where it is created; and, unless it is declared at the outermost level of a module, is automatic (that is, storage for the variable is allocated only during execution of the block in which the variable is declared.)

type

Data type defining the values that the variable can have. Only values within this data type are allowed. Types are described in chapter 4.

initial_value

Initial value assigned to the variable. It can be a constant expression, an indefinite value constructor (described under Initialization later in this chapter), or a pointer to a global procedure. Only a static variable can be assigned an initial value. Initialization is discussed later in this chapter.

This parameter is optional. If omitted, the variable is undefined and filled with the loader's preset value.

Any variable referenced in a program must be declared with the VAR declaration. A variable can be declared only once at each block level although it can be redefined in another block or in a contained (nested) block.

The type assigned to a variable defines the range of values it can take on and also the operations, functions, and procedures that can use it. CYBIL checks to ensure that the operations performed on variables are compatible with their types.

Examples:

The following declarations define a variable named SCORES that can be any integer number, a variable named STATUS that can be either of the boolean values FALSE or TRUE, and two variables named ALPHA1 and ALPHA2 that can be characters.

```
VAR scores : integer;
VAR status : boolean;
VAR alpha1 : char;
VAR alpha2 : char;
```

The declarations for the two character type variables, ALPHA1 and ALPHA2, could be combined as follows:

```
VAR alpha1, alpha2 : char;
```

To combine all of the variables in one declaration, you could use:

```
VAR scores : integer,
  status : boolean,
  alpha1, alpha2 : char;
```

Attributes

Attributes control three characteristics of a variable:

Access - whether the variable can be both read and written

Scope - where within the program the variable can be referenced

Storage - when and where the variable is stored

Access

The access attribute that you can specify is READ. A variable declared with the READ attribute can only be read. It must be initialized in the declaration and cannot be assigned another value later. It is called a read-only variable. If the READ attribute is omitted, CYBIL assumes the variable can be both read and written (changed).

The READ attribute is enforced by software; that is, the compiler checks to ensure that the value of a variable does not change. The READ attribute alone does not mean that the variable is actually in a read-only section.† To do that, you must specify the name of a read-only section as declared in a section declaration (described later in this chapter).

A variable with the READ attribute specified is assumed to be static. (For further information on static variables, refer to Storage later in this chapter.) A read-only variable can be used as an actual parameter in a procedure call only if the corresponding formal parameter is a value parameter; that is, a read-only variable can be passed to a procedure only if the procedure makes no attempt to assign a value to it. (Procedure parameters are described in chapter 7.)

A read-only variable is similar to a constant, but can't always be used in the same places. For example, the initial value that can be assigned to a variable (as described earlier in this chapter) must be a constant expression, an indefinite value constructor, or a pointer to a global procedure. In this case, even though a read-only variable has a constant value, it cannot be used in place of a constant expression. Also, as mentioned in chapter 2, you cannot reference a substring of a constant. You can, however, reference a substring of a variable and, thus, a read-only variable. There are other differences similar to these. The descriptions in this manual state explicitly whether constants and/or variables can be used.

† A read-only section is a hardware feature. Data that resides in a physical area of the machine designated as a read-only section is protected by hardware, not by software. This feature is described in further detail in volume II of the virtual state hardware reference manual.

Examples:

In this example the variable DEBUG is a read-only variable set to the constant value of TRUE. NUMBER can be read and written.

```
VAR
  debug : [READ] boolean := TRUE,
  number : integer;
```

The following example illustrates a difference between constants and read-only variables. To declare a string type, you must specify the length of the string in parentheses following its name. As defined in chapter 4, the length must be a positive, integer constant expression.

```
CONST
  string_size_1 = 5;

VAR
  string_size_2 : [READ] integer := 5,
  string1 : string (string_size_1),
  string2 : string (string_size_2);
```

The declaration of STRING1 is valid; the length of the string is 5 which is the value of the constant STRING_SIZE_1. However, STRING2 is invalid; even though STRING_SIZE_2 does not change in value, it is still a variable and cannot be used in place of a constant expression.

Scope

The scope attributes define the part or parts of a module to which a variable declaration applies. If no scope attributes are included in the declaration, the scope of a variable is the block in which it is declared. A variable declared in an outermost block applies to that block and all the blocks it contains. However, a variable declared even at the outermost level of a module cannot be used outside of that module. The scope attributes, XDCL and XREF, are used to extend the scope of a variable so that it can be shared among modules.

To use the same variable in different modules, you must specify the XDCL and XREF attributes. The XDCL attribute indicates that the variable being declared can be referenced from other modules. The XREF attribute indicates that the variable is declared in another module. When the loader loads modules, it resolves variable declarations so that each XDCL variable is allocated static storage and the XREF variable shares the same space. This is known as satisfying externals. The loader issues an error if an XREF variable does not have a corresponding XDCL variable. In one compilation unit or group of units that will be combined for execution, a specific variable can have only one declaration that contains the XDCL attribute.

Declarations for a shared variable must match except for initialization. A variable declared with the XDCL attribute can be initialized and have different values assigned during program execution. A variable declared with the XREF attribute cannot be initialized but can be assigned values.

The #GATE attribute is an extension of the XDCL attribute. It allows the variable to be accessed by a procedure at a higher ring level.† If #GATE is specified, XDCL must also be specified or a compilation error occurs.

If any attributes are declared, the variable is assumed to be static in storage. If no attributes are declared, the variable is assumed to be automatic, unless it is declared at the outermost level of the module. (A variable declared at the outermost level is always static.)

Example:

Assume the following two modules have been compiled. When the loader loads the resulting object modules and satisfies externals, it allocates storage to FLAG, an XDCL variable, and initializes it to FALSE. When the loader finds the XREF variable FLAG in module TWO, it assigns the same storage. Thus, references to FLAG from either module refer to the same storage location.

```
MODULE one:
    :
  VAR
    flag : [XDCL] boolean := FALSE;
    :
MODEND one;

MODULE two;
    :
  VAR
    flag : [XREF] boolean;
    :
MODEND two;
```

† A ring level is a hardware feature. Rings provide hardware protection in that an unauthorized program cannot access anything at a lower ring level. The ring levels you have access to are determined by your site administrator. For further information on rings, refer to the SCL Object Code Management manual and volume II of the virtual state hardware reference manual.

Storage

The storage attributes determine when storage is allocated and where storage is allocated.

When Storage Is Allocated

There are two methods of allocating storage for variables: automatic and static. For an automatic variable, storage is allocated when the block containing the variable's declaration begins execution. Storage is released when execution of the block ends. If the block is entered again, storage is allocated again, and so on. When storage is released, the value of the variable is lost.

For a static variable, storage is allocated (and initialized, if that parameter is included) only once, at load time. Storage remains allocated throughout execution of the module. However, even though storage remains allocated, a static variable still follows normal scope rules. It can be accessed only within the block in which it is declared. A reference to a static variable from an outer block is an error even though storage for the static variable is still allocated.

The ability to declare a static variable is important, for example, in the case where an XDCL variable is referenced by a procedure before the procedure that declares the variable is executed. Because an XDCL variable is static (refer to Scope earlier in this chapter for further information), it is allocated space and is initialized immediately at load time; therefore, it is available to be referenced before execution of the procedure that actually declares it as XDCL.

A variable can be declared static explicitly with the STATIC attribute. It is assumed to be static implicitly if it is in the outermost level of a module or if it has any other attributes declared. In all other cases, CYBIL assumes the variable is automatic. Only a static variable can be initialized.

The period between the time storage for a variable is allocated and the time that storage is released is called the lifetime of the variable. It is defined in terms of modules and blocks. The lifetime of an automatic variable is the execution of the block in which it is declared. The lifetime of a static variable is the execution of the entire module. An attempt to reference a variable beyond its lifetime causes an error and unpredictable results.

The lifetime of a formal parameter in a procedure is the lifetime of the procedure in which it is a part. Storage space for the parameter is allocated when the procedure is called and released when the procedure finishes executing.

The lifetime of a pointer must be less than or equal to the lifetime of the data to which it is pointing.

The lifetime of a variable that is allocated using the storage management statements (described in chapter 5) is the time between the allocation of storage and the release of storage. A variable allocated by an automatic pointer (using the ALLOCATE statement) must be explicitly freed (using the FREE statement) before the block is left, or the space will not be released by the program. When the block is left, the pointer no longer exists and, therefore, the variable cannot be referenced. If the block is entered again, the previous pointer and the variable referenced by the pointer cannot be reclaimed.

Example:

In this example, the variables COUNTER and FLAG will exist during execution of the entire module; however, they can be accessed only within program MAIN.

```
PROGRAM main;
  VAR
    counter : [STATIC] integer := 0,
    flag : [STATIC] boolean;
       :
PROCEND main;
```

Where Storage Is Allocated

You can optionally specify that storage for a variable be allocated in a particular section. A section is a storage area that can hold variables sharing common access attributes, that is, read-only variables or read/write variables. You define the section and its access attributes yourself using the section declaration (discussed later in this chapter).

If you define a section with the section READ attribute, you define a read-only section in the hardware.† Any variable declared with that section's name as an attribute will reside in that read-only section. When you specify the name of a read-only section in a variable declaration, you must also include the variable access attribute READ.

Example:

This example defines a read-only section named NUMBERS. The variable INPUT_NUMBER is a read-only variable that also resides in the section NUMBERS. In the variable declaration, the READ attribute causes the compiler to check that the variable is not written; the read-only section name, NUMBERS, causes the hardware to ensure that the variable is not written.

```
SECTION
  numbers : READ;
VAR
  input_number : [READ, numbers] integer := 100;
```

† A read-only section is a hardware feature. Data that resides in a physical area of the machine designated as a read-only section is protected by hardware, not by software. This feature is described in further detail in volume II of the virtual state hardware reference manual.

Initialization

An initial value can be assigned to a variable only if it is a static variable. The value can be a constant expression, an indefinite value constructor (described next), or a pointer to a global procedure. The value must be of the proper type and in the proper range. If no initial value is specified, the value of the variable is undefined.

An indefinite value constructor is essentially a list of values. It is used to assign values to the structured types sets, arrays, and records. It allows you to specify several values rather than just one. Values listed in a value constructor are assigned in order (except for sets, which have no order). The types of the values must match the types of the components in the structure to which they are being assigned. An indefinite value constructor has the form

[value {*,value*}...**]**

where value can be one of the following:

- A constant expression.
- Another value constructor (that is, another list).
- The phrase

  REP number OF value

  which indicates the specified value is repeated the specified number of times.
- The asterisk character (*), which indicates the element in the corresponding position is uninitialized.

The REP phrase can be used only in arrays. The asterisk can be used only in arrays and records. For further information, refer to the descriptions of arrays and records in chapter 4.

If an initial value is assigned to a string variable and the variable is longer than the initial value, spaces are appended on the right of the initial value to fill the field. If the initial value is longer than the variable, the initial value is truncated on the right to fit the variable.

In a variant record, fields are initialized in order until a special variable called the tag field name is initialized. The tag field name is then used to determine the variant for the remaining field or fields in the record, and they are likewise initialized in order.

Depending on the attributes defined in the variable declaration, initialization is required, prohibited, or optional. Table 3-1 shows the initialization possible for various attributes.

Table 3-1. Attributes and Initialization

Attributes Specified†	Initialization
None	Optional if static variable; prohibited if automatic variable.
READ	Required.
READ,STATIC	Required.
READ,XDCL	Required.
READ,STATIC,XDCL	Required.
READ,section_name	Required.
READ,XDCL,section_name	Required.
XREF	Prohibited.
XREF,READ	Prohibited.
XREF,STATIC	Prohibited.
XREF,READ,STATIC	Prohibited.
STATIC	Optional.
XDCL	Optional.
XDCL,STATIC	Optional.
section_name	Optional.
section_name,XDCL	Optional.

† The static attribute is assumed if any attributes are specified.

Example:

The variables declared in this example are inside program MAIN. Therefore, they are automatic unless declared with an attribute. TOTAL is automatic and as such cannot be initialized. COUNT is declared static and can be initialized. ALPHA and BETA are also static and can be initialized because they have other attributes declared.

```
PROGRAM main;
     :
  VAR
    total : integer,
    count : [STATIC] integer := 0,
    alpha, beta : [XDCL,READ] char := 'p';
     :
PROCEND main;
```

'ype Declaration

he standard data types that are defined in CYBIL are described in
iapter 4. Any of these can be declared as a valid type within a variable
eclaration. The type declaration allows you to define a new data type
nd give it a name, or redefine an existing type with a new name. Then
iat name can be used as a valid type within a variable declaration.

he format of the type declaration is:

'YPE name = type {,*name* = *type*}...;

name

Name to be given to the new type.

type

Any of the standard types defined by CYBIL or another user-defined type.

)nce you define a type, you can use it to define yet another type. Thus, you
:an build a very complex type that can be referred to by a single name.

'he type declaration is evaluated at compilation time. It does not occupy
itorage space during execution.

Ξxample:

n this example, INT is defined as a type consisting of all the integers; it is
ust a shortened name for a standard type. LETTERS is defined as a type
:onsisting of the characters A through Z only; this is a selective subset of the
itandard type characters. DEVICES is an ordinal type that in turn is used to
lefine EQ_TABLE, a type consisting of an array of 10 elements. Any
ılement in the type EQ_TABLE can have one of the ordinal values specified
n DEVICES.

```
ΓYPE
  int = integer,
  letters = 'a'..'z',
  devices = (lp512, dk844, dk885, nt679),
  eq_table = array [1..10] of devices;
/AR
  i : int,
  alpha : letters,
  table_1 : eq_table,
  status_table : array [1..3] of eq_table;
```

All of the variables in the preceding example could have been declared strictly using variable declarations, as in:

```
VAR
  i : integer,
  alpha : 'a'..'z',
  table_1 : array [1..10] of (lp512, dk844, dk885, nt679),
  status_table : array [1..3] of array [1..10] of
    (lp512, dk844, dk885, nt679);
```

However, it obviously becomes quite cumbersome to declare a complex structure using only standard types. Defining your own types lets you avoid needless repetition and the increased possibility of errors. In addition, it makes code easier to maintain; to add a new device, you need add it only in the type declaration, not in every variable declaration that contains devices.

Section Declaration

A section is an optional working storage area that contains variables with common access attributes. Including the section name in a variable declaration causes the variable to reside in that section.

The format of the section declaration is:

SECTION name {,*name*}... **: attribute**
{,*name* {,*name*}... : *attribute*}...;

name

Name of the section.

attribute

The keyword READ or WRITE.

A section defined with the READ attribute is considered a read-only section.† A variable declared with that section's name will reside in read-only memory. In this case, the variable access attribute READ must also be included in the variable declaration. The section name causes hardware protection; the READ attribute causes compiler checking.

A section defined with the WRITE attribute contains variables that can be both read and written.

The initialization of variables declared with a section name depends on their attributes, as shown in table 3-1. Variables declared with a section name are static.

The SCL Object Code Management manual gives further information on sections regarding the object module format expected as input by the loader and the object library generator.

† A read-only section is a hardware feature. Data that resides in a physical area of the machine designated as a read-only section is protected by hardware, not by software. This feature is described in further detail in volume II of the virtual state hardware reference manual.

Example:

Two sections are defined in this example: LETTERS is a read-only section and NUMBERS is a read/write section. The variable CONTROL_LETTER is a read-only variable that resides in LETTERS. The READ attribute is required because of the read-only section name. UPDATE_NUMBER is a variable that can be read or written, and resides in the section NUMBERS. In this example, it is also declared as an XDCL variable but this is not required.

```
SECTION
  letters : READ,
  numbers : WRITE;
VAR
  control_letter : [READ,letters] char := 'p',
  update_number : [XDCL,numbers] integer;
```

Types 4

This chapter describes the standard types predefined by CYBIL.

Types 4

There are many standard types defined within CYBIL. A variable can be assigned to (that is, an element of) any of these types. The type defines characteristics of the variable and what operations can be performed using the variable. In general, operations involving nonequivalent types are not allowed; one type cannot be used where another type is expected. Exceptions are noted in the descriptions that follow.

In this chapter, types are grouped into three major categories: basic types, structured types, and storage types.

Basic types are the most elementary. They can stand alone but are also used to build the more complex structures. The basic types are:

- Scalar types (integer, character, boolean, ordinal, and subrange)
- Floating-point types (real)
- Cell types
- Pointer types

Structured types are made from combinations of the basic types. The structured types are:

- Strings
- Arrays
- Records
- Sets

Storage types hold groups of components of various types. The storage types are:

- Heaps
- Sequences

Most types, when they are declared, have a fixed size. Strings, arrays, records, sequences, and heaps can also be declared with an adaptable size that is not fixed until execution. For this reason, they are sometimes called adaptable types. Adaptable strings, arrays, records, sequences, and heaps are discussed at the end of this chapter.

Using Types

Types are used as parameters in two kinds of declarations: the variable declaration (to associate a type with a variable name) and the type declaration (to associate a type with a new type name). Both declarations are described in detail in chapter 3, but their basic formats are:

VAR name : { *[attributes]* } **type** { := *initial_value* };

TYPE name = type;

The description of each type shown in this chapter will give the keyword and any additional information necessary to specify that type as a parameter. They replace the generic word "**type**" in the variable and type declarations. For example, the keyword to specify an integer type is INTEGER. The variable declaration would be:

VAR name : { *[attributes]* } **INTEGER** { := *initial_value* };

The type declaration would be:

TYPE name = INTEGER;

Equivalent Types

As mentioned earlier in this chapter, operations involving nonequivalent types are not allowed. Two types can be equivalent, though, even if they don't appear to be identical. For example, two arrays can have different expressions defining their sizes, but the expressions may yield the same value. Rules for determining whether types are equivalent are given in the following descriptions of the types.

Adaptable types and bound variant record types (described under Records later in this chapter) actually define classes of related types that vary by a characteristic, such as size. Adaptable type variables, bound variant record type variables, and pointers to both types are fixed explicitly at execution time. These types are said to be potentially equivalent to any of the types to which they can adapt. That is, during compilation, references to adaptable types and bound variant record types are allowed wherever there is a reference to one of the types to which they can adapt. However, further type checking is done during execution when each type is fixed (assigned to a specific type). It is the current type of an adaptable or bound variant record type that determines what operations are valid for it at any given time.

Basic Types

Scalar Types

All scalar types have an order; that is, for every element of a scalar type you can find its predecessor and successor.

Scalar types are made up of five types:

- Integer
- Character
- Boolean
- Ordinal
- Subrange

Integer

The keyword used to specify an integer type is:

INTEGER

Integers range in value from -(2^{63}-1) through 2^{63}-1.

In general, the subrange type should be used rather than the integer type. This allows the compiler to perform more rigorous type-checking and reduces the amount of storage needed to hold the value.

The following operations are permitted on integers: assignment, addition, subtraction, multiplication, division (both quotient and remainder), all relational operations, and set membership. Refer to Operators in chapter 5 for further information on operations.

The functions $INTEGER and $REAL, described in chapter 6, convert between integer type and real type. The $CHAR function, also described in chapter 6, converts an integer value from 0 through 255 to a character according to its position in the ASCII collating sequence.

Example:

This example shows the definition of a new type named INT, which consists of elements of the type integer. The variable declaration declares variable I to be of type INT, which is the integer type just declared. Also declared as a variable is NUMBERS, which is explicitly of integer type. Because NUMBERS is static, it can be initialized.

```
TYPE
  int = integer;
VAR
  i : int,
  numbers : [STATIC] integer := 100,
```

Character

The keyword used to specify a character type is:

CHAR

An element of the character type can be any of the characters in the ASCII character set defined in appendix B. It is always a single character; more than one character is considered a string. (A string is a structured type that is discussed later in this chapter. A string of length 1 can sometimes be used as a character. Refer to Substrings under Strings later in this chapter.)

The following operations are permitted on characters: assignment, all relational operations, and set membership. A character can be assigned to and compared to a string of length 1. Refer to Operators in chapter 5 for further information on operations and to Strings later in this chapter for further information on string assignment.

The $INTEGER function described in chapter 6 converts a character value to an integer value based on its position in the ASCII collating sequence. The $CHAR function, also in chapter 6, converts an integer value between 0 and 255 to a character in the ASCII collating sequence.

Example:

This example shows the definition of a new type named LETTERS, which consists of elements of the type character. The variable declaration declares variable ALPHA to be of type LETTERS, which is type character; it is static and initialized to the character J. The variable IDS is explicitly declared to be of type character.

```
TYPE
  letters = char;
VAR
  alpha : [STATIC] letters := 'j',
  ids : char;
```

Boolean

The keyword used to specify a boolean type is:

BOOLEAN

An element of the boolean type can have one of two values: FALSE or TRUE. As with other scalar types, boolean values are ordered. Their order is FALSE, TRUE. FALSE is always less than TRUE.

You get a boolean value by performing a relational operation on integers, characters, ordinals, floating-point numbers, or boolean values.

The following operations are permitted on boolean values: assignment, all relational operations, set membership, and boolean sum, product, difference, exclusive OR, and negation. Refer to Operators in chapter 5 for further information on operations.

The $INTEGER function described in chapter 6 converts a boolean value to an integer value. Zero (0) is returned for FALSE; one (1) is returned for TRUE.

Example:

This example shows the definition of a new type named STATUS, which consists of the boolean values FALSE and TRUE. The variable declaration declares variable CONTINUE to be of type STATUS; that is, it can be either FALSE or TRUE. The variable DEBUG is explicitly declared to be boolean and, because it is a read-only variable and therefore static, it can be initialized.

```
TYPE
  status = boolean;
VAR
  continue : status,
  debug : [READ] boolean := TRUE;
```

Ordinal

The ordinal type differs from the other scalar types in that you, the user, define the elements within the type and their order. The term ordinal refers to the list of elements you define; the term ordinal name refers to an individual element within the ordinal.

The format used to specify an ordinal is:

(name, name {*,name...*})

name

Name of an element within the ordinal. There must be at least two ordinal names.

The order is given in ascending order from left to right.

Each ordinal name can be used in just one ordinal type. If a name is used in more than one ordinal, a compilation error occurs.

Ordinals are used to improve the readability and maintainability of programs. They allow you to use meaningful names within a program rather than, for example, map the names to a set of integers that are then used in the program to represent the names.

The following operations are permitted on ordinals: assignment, all relational operations, and set membership.

Two ordinal types are equivalent if they are defined in terms of the same ordinal type names.

The $INTEGER function described in chapter 6 converts an ordinal value (name) to an integer value based on its position within the defined ordinal.

Examples:

In this example, the type declaration defines a type named COLORS, which is an ordinal that consists of the elements RED, GREEN, and BLUE. The variable PRIMARY_COLORS is of COLORS type and therefore has the same elements. The variable WORK_DAYS explicitly declares the ordinal consisting of elements MONDAY through FRIDAY.

```
TYPE
  colors = (red, green, blue);
VAR
  primary_colors : colors,
  work_days : (monday, tuesday, wednesday, thursday,
    friday);
```

In the ordinal type COLORS, the following relationships hold:

RED < GREEN

RED < BLUE

GREEN < BLUE

You can find the predecessor and successor of every element of an ordinal. You can also map each element onto an integer using the $INTEGER function (described in chapter 6.) For example, $INTEGER(RED) = 0; this is the first element of the ordinal.

The type declaration

```
TYPE
  primary_colors = (red, green, blue),
  hot_colors = (red, orange, yellow);
```

is in error because the name RED appears in two ordinal definitions.

Subrange

A subrange is not really a new type but a specified range of values within an existing scalar type. A variable defined by a subrange can take on only the values between and including the specified lower and upper bounds.

The format used to specify a subrange is:

lowerbound .. upperbound

lowerbound
Scalar expression specifying the lower bound of the subrange.

upperbound
Scalar expression specifying the upper bound of the subrange.

The lower bound must be less than or equal to the upper bound. Both bounds must be of the same scalar type.

The type of a subrange is the type of its lower and upper bounds. If a subrange completely encompasses its own type, it is said to be an improper subrange type. For example, the subrange

FALSE..TRUE

is of type boolean and also contains every element of type boolean. It is equivalent to specifying the type itself. An improper subrange type is always equivalent to its own type.

Two subranges are equivalent if they have the same lower and upper bounds.

Subranges allow for additional error checking. Compilation options are available that cause the compiler to check assignments during program execution and issue an error if it finds a variable not within range. (Range checking is available as an option on the compiler call command and as a compiler directive. They are both described in chapter 8.) In addition, subranges improve readability. Because a subrange defines the valid range of values for a variable, it is more meaningful to the user for documentation and maintenance.

The operations permitted on a subrange are the same as those permitted on its type (the type of its lower and upper bound).

Example:

This example shows the definition of a new type named LETTERS, which consists of the characters A through Z only. It also defines an ordinal named COLORS consisting of the colors listed. The variable declaration declares variable SCORES to consist of the numbers 0 through 100. The lower and upper bounds are of integer type, so the subrange is also an integer type. STATUS is a subrange of boolean values, which could have been declared simply as BOOLEAN. HOT_COLORS is a subrange of the ordinal type COLORS. It consists of the colors RED, ORANGE, and YELLOW.

```
TYPE
  letters = 'a'..'z',
  colors = (red, orange, yellow, white, green, blue);
VAR
  scores = 0..100,
  status = FALSE..TRUE,
  hot_colors = red..yellow;
```

Floating-Point Type

The floating-point type defines real numbers.

Real

The keyword used to specify a real type is:

REAL

Real numbers range in value from $4.8*10^{-1234}$ through $5.2*10^{1232}$.

The following operations are permitted on real types: assignment, addition, subtraction, multiplication, division, and all relational operations.

The functions $INTEGER and $REAL, described in chapter 6, convert between integer type and real type.

Cell Type

The cell type represents the smallest storage location that is directly addressable by a pointer. On NOS/VE, a cell is an 8-bit byte within a 64-bit memory word.

The keyword used for specifying a cell type is:

CELL

Operations permitted on a cell type are assignment and comparison for equality and inequality.

Pointer Types

A pointer represents the location of a value rather than the value itself. When you reference a pointer, you indirectly reference the object to which it is pointing.

The format for specifying a pointer type is:

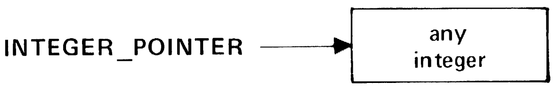
^ type

type
Type to which the pointer can point. It can be any defined type. With the exception of a pointer to cell type (discussed later in this chapter), the pointer can point only to objects of the type specified.

For example,

```
VAR integer_pointer = ^integer;
```

defines a pointer named INTEGER_POINTER that can point only to integers.



The format for specifying the object of a pointer (that is, what the pointer points to) is:

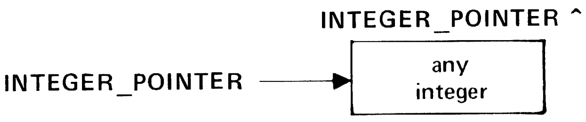
pointer_name ^

pointer_name
The name you gave the pointer in the variable declaration.

This preceding notation is called a pointer reference; it refers to the object to which pointer_name points. It can also be referred to as a dereference. For example,

```
integer_pointer ^
```

identifies a location in memory; it is the location to which INTEGER_ POINTER points.



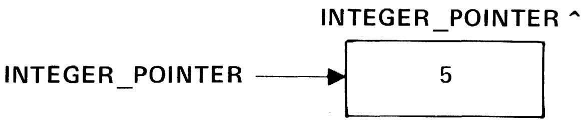
You can initialize or assign a value to the object of a pointer as you would any other variable; that is:

pointer_name ^ := value;

This assigns the specified value to the object that the pointer points to. For example,

```
integer_pointer ^ := 5;
```

assigns the integer value 5 to the location INTEGER_POINTER points to:



You can assign the object of a pointer to a variable in the same way:

variable := pointer_name ^;

This takes the value of what pointer_name points to and assigns it to the variable. For example,

```
i := integer_pointer ^;
```

assigns to I the contents of what INTEGER_POINTER points to, that is, 5.

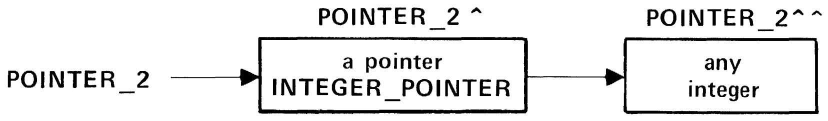
If a pointer reference is to another pointer type variable, meaning that the pointer points to a pointer that in turn points to a variable, you can specify the variable with the form:

pointer_name ^^

For example, the declarations

```
TYPE
  integer_pointer = ^integer;
VAR
  pointer_2: ^integer_pointer;
```

can be pictured conceptually as follows:



POINTER_2 points to a pointer of type INTEGER_POINTER. INTEGER_POINTER points to integers. A reference to POINTER_2 ^ refers to the location of the pointer that in turn points to an integer. A reference to POINTER_2 ^^ refers to the location of the integer.

The value assigned to a pointer can be:

- The pointer constant NIL.
- The pointer symbol ^ followed by a variable of the type to which the pointer can point.
- A pointer variable.
- A pointer-valued function.

NIL is the value of a pointer variable without an object; the variable is not currently assigned to any location. It can be assigned to or compared with any pointer of any type.

Pointers allow you to manipulate storage dynamically. Using pointers, you can create and destroy variables while a program is executing. Memory is allocated when the variable is created and released when it is destroyed. Pointers also allow you to reference the variables without giving each a unique name.

A pointer variable can be a component of a structured type as well as a valid parameter in a function. A function can return a pointer variable as a value.

Permissible operations on pointers are assignment and comparison for equality and inequality.

Pointers to adaptable types (adaptable strings, arrays, records, sequences, and heaps) provide the only method for accessing objects of these types other than through formal parameters of a procedure. In particular, pointers to adaptable types and pointers to bound variant records are used to access adaptable variables and bound variant records whose types have been fixed by an ALLOCATE, PUSH, or NEXT statement (described in chapter 5).

Pointers are equivalent if they are defined in terms of equivalent types. A pointer to a fixed type (as opposed to an adaptable type) can be assigned and compared to a pointer to an adaptable type or bound variant record if the adaptable type is potentially equivalent to the fixed type. (Refer to Equivalent Types earlier in this chapter for further information on potentially equivalent types.)

Example:

The following example shows the declaration and manipulation of two pointer type variables. Comments appear to the right.

```
TYPE ptr = ^integer;          PTR is a type that can contain pointers to
                              integers.
VAR i, j, k : integer,
  p1 : ptr,                   P1 is a variable that can contain pointers to
                              integers.
  p2 : ^p1,                   P2 is a variable that can contain pointers to
                              P1 (that is, pointers that point to pointers to
                              integers). It could have been written as
                              P2 : ^^ INTEGER.
  b1, b2 : boolean;
ALLOCATE p1;                  Allocates space for an integer (because that
                              is what P1 points to) and sets P1 to point to
                              that space.
ALLOCATE p2;                  Allocates space for a pointer that points to
                              an integer and sets P2 to point to that
                              pointer.
p1^ := 10;                    The space pointed to by P1 is set to 10.
p2^ := p1;                    The space pointed to by P2 is set to the
                              value of the pointer P1.
j := p1^;                     The integer variable J is set to what P1
                              points to: the integer 10.
k := p2^^;                    The integer variable K is set to the object of
                              the pointer that P2 points to. (Think of P2 ^^
                              as "P2 points to a pointer; that pointer
                              points to an object." You are assigning that
                              object to K.) P2 points to P1, which points to
                              the integer 10.
b1 := j = k;                  J and K are both 10. B1 is TRUE.
b2 := p1^ = p2^^;             P1 points to an integer. P2 points to the
                              pointer (P1) that points to the same integer.
                              Their values are the same and B2 is TRUE.
p1 := NIL;                    P1 no longer points to anything.
k := p1^;                     The statement is in error because P1 does
                              not point to anything.
IF p2 = NIL THEN              A valid statement. K is not incremented
  k := k + 1                  because P2 still points to P1.
IFEND;
p1 := ^(i + j + 2 * k);       An invalid statement. The location of an
                              expression cannot be found.
```

Pointer to Cell

A pointer to cell type can take on values of any type.

The format for declaring a pointer to a cell is:

```
^CELL
```

A variable declared simply as a pointer type variable can take on as values only pointers to a single type, which is specified in the pointer's declaration. A variable declared as a pointer to cell variable has no such restrictions. It can take on values of any type. Also, any fixed or bound variant pointer variable can assume a value of pointer to cell.

Permissible operations on a pointer to a cell are assignment and comparison for equality and inequality. In addition, a pointer to a cell can be assigned to any pointer to a fixed or bound variant type. But the pointer to the fixed or bound variant type cannot have as its value a pointer to a variable that is not a cell type or, furthermore, whose type is not equivalent to the type to which the target of the assignment points. A pointer to a cell can be the target of assignment of any pointer to a fixed or bound variant type.

Relative Pointer

Relative pointer types represent relative locations of components within an object with respect to the beginning of the object.

The format for specifying a relative pointer is:

REL { *(parent_name)* } **^component_type**

parent_name

Name of the variable that contains the components being designated by relative pointers. The variable can be a string, array, record, heap, or sequence type (either fixed or adaptable). If omitted, the default heap is used.

component_type

Type of the component to which the relative pointer will point.

Relative pointers are generated using the standard function #REL (described in chapter 6). A relative pointer cannot be used to access data directly. Instead, the relative pointer must be converted to a direct pointer using the standard function #PTR (also described in chapter 6). The direct pointer can then be used to access the data.

Relative pointers have three major differences from the other pointers discussed in this chapter:

- Relative pointers may need less space than other pointers.
- A linked list or array of relative pointers (or some similar organization) within a parent type variable is still correct if the entire variable is assigned to another variable of the same parent type.
- Relative pointers are independent of the base address of the parent type variable.

Operations permitted on a relative pointer are assignment, comparison for equality and inequality, and the #PTR function. Relative pointers can be assigned and compared if they are of equivalent relative pointer types. Relative pointer types are equivalent if they are defined in terms of equivalent parent types and equivalent component types.

Structured Types

Structured types are combinations of the basic types already described in this chapter (integer, character, boolean, ordinal, subrange, real, pointer, and cell). Even the structured types discussed here can be combined with each other but they are still essentially groups of the basic types. The structured types described in this section are:

- Strings
- Arrays
- Records
- Sets

Strings

A string is one or more characters that can be identified and referenced as a whole by one name.

The format used to specify a string type is:

STRING (length)

length
A positive integer constant expression from 1 through 65,535.

If an initial value is specified in the variable declaration for a string, it can be:

- A string constant.
- The name of a string constant declared with a constant declaration.
- A constant expression (as described in chapter 2).

A string cannot be packed. Two string types are equivalent if they have the same length.

The following operations are permitted on string types: assignment and comparison (all six relational operations). For further information, refer to Assigning and Comparing String Elements later in this chapter.

Substrings

You can reference a part of a string (this is called a substring) or a single character of a string.

The format for referencing a substring or single character is:

name (position {, *length*})

name

Name of the string.

position

Position within the string of the first character of the substring. (The position of the first character of the string is always 1.) It must be a positive integer expression less than or equal to the length of the string plus one; that is,

$1 \leq \text{position} \leq \text{string length} + 1$

If the string length plus one is specified, the substring is an empty string.

length

Number of characters in the substring. It must be a non-negative integer expression or * (the asterisk character). If * is specified, the substring consists of the character specified by the position parameter and all characters following it in the string. If 0 is specified, the substring is an empty string. If the parameter is omitted, a length of 1 is assumed.

A substring reference in the form

name(position)

is a substring of length 1, a single character. In this form, it can be used anywhere a character expression is allowed. It can be:

- Compared with a character.
- Tested for membership in a set of characters.
- Used as the initial and/or final value in a FOR statement that is controlled by a character variable.
- Used as a value in a CASE statement.
- Used as an argument in the standard functions $INTEGER, SUCC, and PRED.
- Assigned to a character variable.
- Used as an actual parameter to a formal parameter of type character.
- Used as an index value corresponding to a character type index in an array.

A string constant, even if it is declared with a name in a CONST declaration, is not a variable. Therefore, substrings cannot be referenced in a string constant.

Examples:

If a string variable LETTERS is declared and initialized as follows

```
VAR letters : string (6) := 'abcdef';
```

the following substring references are valid:

Substring	Comments
LETTERS(1)	Refers to 'A'.
LETTERS(6)	Refers to 'F'.
LETTERS(1,6)	Refers to the entire string.
LETTERS(1,*)	Refers to the entire string.
LETTERS(2,5)	Refers to 'BCDEF'.
LETTERS(2,*)	Refers to 'BCDEF'.
LETTERS(2,0)	Refers to an empty string ''.
LETTERS(7,*)	Refers to an empty string ''.

LETTERS(0), LETTERS(8) and LETTERS(8,0) are illegal.

If a pointer variable is declared and initialized as follows

```
VAR string_ptr : ^string (6) := ^letters;
```

then STRING_PTR points to the string LETTERS and the pointer variable STRING_PTR^ can be used to make substring references just like the variable LETTERS.

Substring	Comments
STRING_PTR^(1)	Refers to 'A'.
STRING_PTR^(6)	Refers to 'F'.
STRING_PTR^(1,6)	Refers to the entire string.
STRING_PTR^(2,*)	Refers to 'BCDEF'.
STRING_PTR^(2,0)	Refers to an empty string ''.

Assigning and Comparing String Elements

You can assign or compare a character, substring, or string to a substring, string variable, or character variable. A character is treated as a string of length 1.

If you are assigning a value that is longer than the substring or variable to which it is being assigned, the value is truncated on the right. If you are assigning a value that is shorter, spaces are appended on the right to fill the field. This method is also used for comparing strings of different lengths.

If a substring is being assigned to a substring of the same variable, the fields cannot overlap or the results are undefined.

The concatentation operation, CAT, cannot be used with string variables.

Example:

Assume the string variable DAY is declared and initialized as follows:

```
VAR
  day : string (6) := 'monday';
```

The following assignments can be made:

```
short := day (1,3);
empty := day (1,0);
```

SHORT is assigned the string 'MON'. EMPTY is assigned a null string.

Arrays

An array in CYBIL is a collection of data of the same type. You can access an array as a whole, using a single name, or you can access its elements individually.

The format used for specifying an array type is:

{*PACKED*} **ARRAY [subscript_bounds] OF type**

PACKED

Optional packing parameter. When specified, the elements of the array are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If omitted, the array is unpacked; that is, the elements are mapped in storage to optimize access time rather than to conserve space. (The array itself is always mapped into an addressable memory location; that is, it starts on a word boundary or, in the case of a packed array in a record, on a byte boundary.) For further information on how data is stored in memory, refer to appendix D, Data Representation in Memory.

If the array contains structured types (such as records), the elements of that type (the fields in the records) are not automatically packed. The structured type itself must be declared packed.

subscript_bounds

Value that specifies the size of the array and what values you can use to refer to individual elements. The bounds can be any scalar type or subrange of a scalar type, except REAL; the bounds is often a subrange of integers.

type

Type of the elements within the array. The type can be any defined type, including another array, except an adaptable type (that is, an adaptable string, array, or record). All elements must be of the same type.

Elements of a packed array cannot be passed as reference (that is, VAR) parameters in programs, functions, or procedures.

Two array types are equivalent if they have the same packing attribute, equivalent subscript bounds, and equivalent component types.

The only operation permitted on an array type is assignment.

Initializing Elements

An array can be initialized using an indefinite value constructor. An indefinite value constuctor is a list of values assigned in order to the elements of an array. The first value in the list is assigned to the first element, and so on. The number of values in the value constructor must be the same as the number of elements in the array. The type of the values must match the type of the elements in the array. An indefinite value constructor has the form

[value {,*value*}...**]**

where value can be one of the following:

- A constant expression.
- Another value constructor (that is, another list).
- The phrase

  REP number OF value

  which indicates the specified value is repeated the specified number of times.
- The asterisk character (*), which indicates the element in the corresponding position is uninitialized.

An indefinite value constructor can be used only for initialization; it cannot be used to assign values during program execution. Individual elements can be assigned during execution using the assignment statement (described in chapter 5).

Referencing Elements

The array name alone refers to the entire structure. The format for referring to an individual element of an array is:

array_name[subscript]

subscript

A scalar expression within the range and of the type specified in the subscript_bounds field of the array declaration. This subscript specifies a particular element.

Examples:

This example shows the definition of a type named POS_TABLE, which is an array of 10 elements that can take on the values defined in POSITION. The variable declaration declares variable NUMBERS to be an array of five elements initialized to the values 1, 2, 3, 4, and 5 where 1 is the value of the first element, and so on. LETTERS is an array of 26 elements that can be any characters. BIG_TABLE is a 100-element array, each element of which is an array of 10 elements.

```
TYPE
  position = (boi, asis, eoi),
  pos_table = array [1..10] of position;
VAR
  i : integer := 5,
  numbers : array [1..5] of integer := [1,2,3,4,5],
  letters = array ['a'..'z'] of char,
  big_table = array [1..100] of pos_table;
```

The declaration of BIG_TABLE is equivalent to:

```
VAR big_table = array [1..100] of array [1..10] of position;
```

Individual elements can be referenced using the following statements.

`numbers [i]`	This reference is the same as NUMBERS[5]; it refers to the fifth element of the array NUMBERS.
`letters ['b'] := 'B';`	This statement sets the second element of the array LETTERS to the uppercase character B.
`big_table [13][10] := asis;`	This statement sets the tenth element of the thirteenth array to ASIS.

The following example shows the declaration and initialization of a two-dimensional array named DATA_TABLE. All of the components of the third element of the array (which is an array itself) are set to zero. Notice that the third element of the last array, DATA_TABLE [4][3], is uninitialized.

```
TYPE
  innerarray = array [1..5] of integer,
  twodim = array [1..4] of innerarray;

VAR
  data_table : twodim := [ [ 5, -10, 2, 6, 3 ],
                           [ 4, 11, 19, -3, 6],
                           [rep 5 of 0],
                           [ 3, -9, *, 4, 15] ];
```

Records

Records are collections of data that can be of different types. You can access a record as a whole using a single name, or you can access elements individually.

A record has a fixed number of components, usually called fields, each with its own unique name. Different fields are used to indicate different data types or purposes.

There are two types of records: invariant records and variant records. Invariant records consist of fields that don't change in size or type. Variant records can contain fields that vary depending on the value of a key variable. Formats used for specifying both kinds of records are given later in this chapter.

Operations permitted on record types are assignment and, for invariant records only, comparison for equality and inequality. The invariant records being compared cannot contain arrays as fields.

Invariant Records

An invariant record consists of fields that do not vary in size or type once they have been declared. They are called fixed or invariant fields.

The format used for specifying an invariant record is:

{*PACKED*} **RECORD**
  field_name : {*ALIGNED* {*[offset MOD base]*}} **type**
  {*,field_name* : {*ALIGNED* {*[offset MOD base]*}} *type*}...
RECEND

PACKED

Optional packing parameter. When specified, the fields of a record are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If omitted, the record is unpacked; that is, the fields are mapped in storage to optimize access time rather than to conserve space. For further information on how data is stored in memory, refer to appendix D, Data Representation in Memory.

If one of the fields is a structured type (such as another record), the elements of that type are not packed automatically. The structured type itself must be declared packed.

field_name

Name identifying a particular field. The name must be unique within the record. Outside of the record declaration, it can be redefined.

ALIGNED

Optional alignment parameter. If specified, it can appear alone or with an offset, in the form:

ALIGNED [offset MOD base]

When a field is aligned, it is mapped in storage so that it is directly addressable. This means the field begins on an addressable boundary to facilitate rapid access to the field. This may negate some of the effect of packing the record. For further information, refer to Alignment later in this chapter.

offset MOD base

Optional offset to be used in conjunction with the ALIGNED parameter. This offset causes the field to be mapped to a particular hardware address relative to the specified base and offset. It can be a particular word or a particular byte within a word. Base is evaluated first to find the word boundary; offset is then evaluated to determine the number of bytes offset within that word. Filler is created if necessary to ensure that the field begins on the specified word or byte.

offset

Byte offset within the word specified by base. It must be an integer constant less than base.

base

Word boundary. It must be an integer constant that is divisible by 8. For automatic variables, the base can only be 8.

type

Any defined type, including another record, but not an adaptable type.

Elements of a packed record cannot be passed as reference (that is, VAR) parameters in programs, functions, or procedures unless they are aligned.

The only operations possible on whole invariant records are assignment and comparison. A record can be assigned to another record if they are both of the same type. A record can also be compared to another record for equality or inequality if they are both of the same type. Invariant record types are the same if they have the same packing attributes, the same number of fields, and corresponding fields have the same field names, same alignment attribute, and equivalent types.

Example:

This example shows the definition of two new types, both records. The record named DATE has three fields that can hold, respectively, DAY, MONTH, and YEAR. The record named RECEIPTS appears to contain two fields, NAME and PAYMENT; but PAYMENT is itself a record consisting of the three fields in DATE, just described. Initialization of fields within records is discussed under Initializing Elements later in this chapter.

```
TYPE
  date = RECORD
           day : 1..31,
           month : string (4),
           year : 1900..2100,
         RECEND,

  receipts = RECORD
               name : string (40),
               payment : date,
             RECEND;
```

Variant Records

A variant record contains fields that may vary in size, type, or number depending on the value of an optional tag field. These different fields are called variant fields or simply variants.

The format used for specifying a variant record is:

{*PACKED*} {*BOUND*} **RECORD**
  {*fixed_field_name* : {*ALIGNED* {[*offset MOD base*]} *type*}...†
  CASE {*tag_field_name* : } **tag_field_type OF**
  = tag_field_value =
    variant_field
  {= *tag_field_value* =
    variant_field}...
  CASEND
RECEND

† When more than one fixed field is specified, they must be separated by commas.

PACKED

Optional packing parameter. When specified, the fields of a record are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If omitted, the record is unpacked; that is, the fields are mapped in storage to optimize access time rather than to conserve space. For further information on how data is stored in memory, refer to appendix D, Data Representation in Memory.

If a field is a structured type (such as another record), the elements of that type are not packed automatically. The structured type itself must be declared packed.

BOUND

Optional parameter indicating that this is a bound variant record. If specified, the tag_field_name parameter is required. Additional information on bound variant records follows the parameter descriptions.

fixed_field_name

The name of a fixed field (one that does not vary in size), as described under Invariant Records earlier in this chapter. The name must be unique within the record. Outside of the record declaration, it can be redefined. There can be zero or more fixed fields.

ALIGNED

Optional alignment parameter; the same as that for an invariant record. If specified, it can appear alone or with an offset in the form:

ALIGNED [offset MOD base]

When a field is aligned, it is mapped in storage so that it is directly addressable. This means the field begins on an addressable boundary to facilitate rapid access to the field. This may negate some of the effect of packing the record. For further information, refer to Alignment later in this chapter.

offset MOD base

Optional offset to be used in conjunction with the ALIGNED parameter, the same as that for an invariant record. This offset causes the field to be mapped to a particular hardware address relative to the specified base and offset. It can be a particular word or a particular byte within a word. Base is evaluated first to find the word boundary; offset is then evaluated to determine the number of bytes offset within that word. Filler is created if necessary to ensure that the field begins on the specified word or byte.

offset

Byte offset within the word specified by base. It must be an integer constant less than base.

base

Word boundary. It must be an integer constant that is divisible by 8. For automatic variables, the base can only be 8.

type

Any defined type, including another record, but not an adaptable type.

tag_field_name

Optional parameter specifying the name of the variable that determines the variant. The current value of this variable determines which of the variant fields that follow will actually be used. If omitted, the variant that had the last assignment made to one of its fields is used. This parameter is required if the record is a bound variant record (BOUND is specified). Additional information is given following the parameter descriptions.

tag_field_type

Any scalar type. This type defines the values that the tag_field_value can have.

tag_field_value

A constant scalar expression or subrange. It must be one of the possible values that can be assigned to the variable specified by tag_field_name. It must be of the type and within the range specified by tag_field_type. Specifying a subrange has the same effect as listing each value separately.

variant_field

Zero or more fixed fields of the same form as that shown in the second line of this format. This field exists only if the current value of tag_field_name is the same as that in the tag_field_value associated with the variant_field. The last field can be a variant itself.

The variant fields must follow all invariant (fixed) fields in the record. The field following the reserved word CASE is called the tag_field_name. The tag_field_name can take on different values during execution. When its value matches one of the values specified in a tag_field_value, the variants associated with that tag_field_value are used. Variants themselves consists of zero or more fixed fields optionally followed by another variant. If the last field is itself a variant, it can have another CASE clause, tag_field_name, and so on.

The tag_field_name is an optional field. When it is omitted, no storage is assigned for the tag field. If the record has no tag field, you choose a variant by making an assignment to a subfield within a variant. The variant containing that subfield becomes the currently active variant. In a variant record without a tag field, all fields in a new active variant become undefined except the subfield that was just assigned. An attempt to access a variant field that is not currently active produces undefined results.

Space for a variant record is allocated using the largest possible variant.

Variant record types are equivalent if they have the same packing attribute, their fixed fields are equivalent (as defined for invariant record types), they have the same tag field names, their tag field types are equivalent, their tag field values are the same, and their corresponding variant fields are equivalent.

A bound variant record is specified by including the BOUND parameter; the tag_field_name is also required. A bound variant record type can be used only to define pointers for bound variant record types (that is, bound variant pointers). A variable of this type is always allocated in a sequence or heap, or in the run-time stack managed by the system.

When allocating a bound variant record, you must specify the tag field values that select the variation of the record. Only the specified space is allocated. The ALLOCATE statement in this case returns a bound variant pointer.

If a formal parameter of a procedure is a variant record type, the actual parameter cannot be a bound variant record type.

A record cannot be assigned to a variable of bound variant record type.

Bound variant record types are equivalent if they are defined in terms of equivalent, unbound records. A bound variant record type is never equivalent to a variant record type.

Example:

This example defines a type named SHAPE, which becomes the type of the tag field, in this case a variable named S. When S is equal to TRIANGLE, the record containing fields SIZE, INCLINATION, ANGLE1, and ANGLE2 is used as if it were the only record available. When the value of S changes, the record variant being used changes too.

```
TYPE
  shape = (triangle, rectangle, circle),
  angle = -180..180,
  figure = RECORD
             x,
             y,
             area : real,
           CASE s : shape OF
           = triangle =
             size : real,
             inclination,
             angle1,
             angle2 : angle,
           = rectangle =
             side1,
             side2 : integer,
             skew,
             angle3 : angle,
           = circle =
             diameter : integer,
           CASEND,
           RECEND;
```

Initializing Elements

A record can be initialized using an indefinite value constructor. An indefinite value constructor is a list of values assigned in order to the fields of a record. The first value in the list is assigned to the first field, or first element in a field, and so on. The type of the values must match the type of the elements in the field. An indefinite value constructor has the form

[value {,*value*}...**]**

where value can be one of the following:

- A constant expression.
- Another value constructor (that is, another list).
- The asterisk character (*), which indicates the element in the corresponding position is uninitialized.

An indefinite value constructor can be used only for initialization; it cannot be used to assign values during program execution. Individual fields can be assigned during execution using the assignment statement (described in chapter 5).

Example:

The variable BIRTH_DAY, in this example, is a record with the fields described in the record type named DATE. It is initialized using an indefinite value constructor to the 24th day of August, 1950.

```
TYPE
  date = RECORD
           day : 1..31,
           month : string (4),
           year : 1900..2100,
         RECEND;
VAR
  birth_day : date :=[24, AUG, 1950];
```

Referencing Elements

The record name alone refers to the entire structure. The format for accessing a field in a record is:

record_name.field_name {.*sub_field_name*}...

record_name

Name of the record as declared in the variable declaration.

field_name

Name of the field to be accessed. If the field is an array, a reference to an individual element can also be included using the form:

field_name[subscript]

sub_field_name

Optional field name. This parameter is used if the field previously specified is itself a structured type, for example, another record. If the contained field is an array, a reference to an individual element can be included using the form:

sub_field_name[subscript]

Example:

The variable PROFILE is a record with the fields described in the record type STATS. In this example, PROFILE is initialized with the values in the indefinite value constructor in the variable declaration.

```
TYPE stats = RECORD
               age : 6..66,
               married : boolean,
               date : RECORD
                        day : 1..31,
                        month : 1..12,
                        year : 80..90,
                      RECEND,
             RECEND;

VAR profile : stats := [23,FALSE,[3,5,82]];
```

The following references can be made to fields.

`profile.age`	This field contains 23.
`profile.married`	This field contains FALSE.
`profile.date.day`	This field contains 3.
`profile.date.month`	This field contains 5.
`profile.date.year`	This field contains 82.

Alignment

Unpacked records and their fields are always aligned (that is, directly addressable). Even if it is packed, a record itself is always aligned (that is, the first field is directly addressable) unless it is an unaligned field within another packed structure. Fields in a packed record, however, are not aligned unless the ALIGNED attribute is explicitly included. Aligning the first field of a record aligns the entire record.

Unpacked records and their fields, because they are aligned, can always be passed as reference (that is, VAR) parameters in programs, functions, and procedures. Packed records must be aligned to be valid as reference parameters. Packed, unaligned records cannot be used.

Sets

A set is a collection of elements that, unlike arrays and records, is always operated on as a single unit. Individual elements are never referenced.

The format used to specify a set type is:

SET OF scalar_type

scalar_type

Type of all the elements that will be within the set. It can be a scalar type or a subrange of a scalar type.

All members of a set must be of the same type. Members within a set have no specific order; that is, order has no effect in any of the operations performed on sets.

Set types are equivalent if their elements have equivalent types.

Permissible operations on sets are assignment, intersection, union, difference, symmetric difference, negation, inclusion, identity, and membership. Refer to Operators in chapter 5 for further information on set operations. The SUCC and PRED functions are not defined for set types.

The difference (-) or symmetric difference (XOR) of two identical sets is the empty set. The empty set is contained in any set. For a given set, the complement of the empty set, -[ ], is the full set.

Initializing and Assigning Elements

Values can be assigned to a set using an indefinite value constructor or a set value constructor. An indefinite value constructor can be used only for initialization; a set value constructor can be used for both initialization and assignment during program execution.

An indefinite value constructor is a list of values assigned to the set. The type of the values must match the type of the set. An indefinite value constructor has the form:

[value {,*value*}...**]**

value

A constant expression or another indefinite value constructor (that is, another list).

A set value constructor constructs a set through explicit assignment. A set value constructor has the form:

$name [{ *value* {,*value*}...}]

name

Name of the set type. The dollar sign ($) must precede the name to indicate a set value constructor.

value

An expression of the same type as that specified for the set. When used in initialization, only constants or constant expressions are valid. The empty set can be specified by [ ].

A set value constructor can be used wherever an expression can be used.

Example:

This example shows the declaration of a variable named ODD that is a type of a set of integers from 0 through 10. It is initialized with an indefinite value constructor assigning the integers 1, 3, and 5 to the set. The variable VOWELS is a set that can contain any of the letters A through Z. It is assigned the letters A, E, I, O, and U using a set value constructor. It constructs a set of type C, which contains the specified letters; then that set is assigned to the set VOWELS. The variables LIST_1 and LIST_2 are sets that can contain any characters. LIST_1 is assigned, using a set value constructor, the letters X, Y, and Z. LIST_2 is assigned the complement of X, Y, and Z, that is, a set consisting of every character except the letters X, Y, and Z.

```
TYPE
  a = set of 0..10,
  c = set of 'a'..'z',
  ch = set of char;
VAR
  odd : a := [1, 3, 5],
  vowels : c,
  list_1, list_2 : ch;
  .
  .
  .
vowels := $c['a', 'e', 'i', 'o', 'u'];
list_1 := $ch['x', 'y', 'z'];
list_2 := -$ch['x', 'y', 'z'];
```

Storage Types

Storage types represent structures to which variables can be added, deleted, and referenced under program control. (The statements used to access the storage types are described under Storage Management Statements in chapter 5.) There are two storage types:

- Sequences
- Heaps

Sequences

A sequence type is a storage structure whose components are referenced sequentially using pointers. These pointers are constructed by the NEXT and RESET statements (described in chapter 5).

The format used for specifying a sequence type is:

SEQ ({*REP number OF*} **type** {,{*REP number of*} *type*}...)

number

Positive integer constant expression. This is an optional parameter specifying the number of repetitions of the specified type.

type

A fixed type that can be a user-defined type name; one of the predefined types integer, character, boolean, real, or cell; or a structured type using the preceding types.

The phrase "REP number OF type" can be repeated as many times as desired. It specifies that storage must be available to hold the indicated number of occurrences of the named types simultaneously. The types that are actually stored in a sequence do not have to be the same as the types specified in the declaration, but adequate space must have been allocated to hold those types in the declaration. In other words, if a sequence is declared with several repetitions of integer type, the space to hold these integers has to be available, but it might actually hold strings or boolean values.

Sequence types are equivalent if they have the same number of REP phrases and corresponding phrases are equivalent. Two REP phrases are equivalent if they have the same number of repetitions of equivalent types.

Assignment to another sequence is the only operation permitted on sequences.

Heaps

A heap type is a storage structure whose components are allocated explicitly by the ALLOCATE statement and released by the FREE and RESET statements (described in chapter 5). They are referenced by pointers constructed by the ALLOCATE statement.

The format used for specifying a heap type is:

HEAP ({*REP number OF*} **type** {,{*REP number of*} *type*}...)

number

Positive integer constant expression. This is an optional parameter specifying the number of repetitions of the specified type.

type

A fixed type that can be a user-defined type name; one of the predefined types integer, character, boolean, real, or cell; or a structured type using the preceding types.

The phrase "REP number OF type" can be repeated as many times as desired. It specifies that storage must be available to hold the indicated number of occurrences of the named types simultaneously. The types that are actually stored in a heap do not have to be the same as the types specified in the declaration, but adequate space must have been allocated to hold those types in the declaration. In other words, if a heap is declared with several repetitions of integer type, the space to hold these integers has to be available, but it might actually hold strings or boolean values.

Heap types are equivalent if they have the same number of REP phrases and corresponding phrases are equivalent. Two REP phrases are equivalent if they have the same number of repetitions of equivalent types.

The default heap can be managed with the ALLOCATE and FREE statements in the same way as a user-defined heap. For further information, refer to the descriptions of these statements in chapter 5.

Adaptable Types

An adaptable type is a type that has indefinite size or bounds; it adapts to data of the same type but of different sizes and bounds. The types described thus far in this chapter are fixed types. An adaptable type differs from a fixed type in that the storage required for a fixed type is constant and can be determined before execution. Storage for an adaptable type is determined during program execution.

An adaptable type can be a string, array, record, sequence, or heap. An adaptable type can be used to define formal parameters in a procedure and adapatable pointers. Pointers are the mechanism used for referencing adaptable variables.

The size of an adaptable type must be fixed during execution. This can be done in one of three ways:

- If the adaptable type is a formal parameter to a procedure or function, the size is fixed by the actual parameters when the procedure or function is called. You can determine the length of an actual parameter string using the STRLENGTH function, and the bounds of an actual parameter array using the UPPERBOUND and LOWERBOUND functions. (For further information, refer to the description of the appropriate function in chapter 6.)
- An adaptable pointer type on the left side of an assignment statement is fixed by the assignment operation. It can be assigned any pointer whose current type is one of the types that the adaptable type can take on.
- An adaptable type can be fixed explicitly using the storage management statements (described in chapter 5).

An adaptable type is declared with an asterisk taking the place of the size or bounds normally found in the type or variable declaration.

Adaptable Strings

The format used for specifying an adaptable string is:

STRING (* {<= *length*})

length

Optional parameter specifying the maximum length of the adaptable string. If omitted, 65,535 characters is assumed.

If the string exceeds the maximum allowable length, an error occurs.

Two adaptable string types are always equivalent.

Adaptable Arrays

The format used for specifying an adaptable array is:

{*PACKED*} **ARRAY [**{*lower_bound* ..} ***] OF type**

PACKED

Optional packing parameter. When specified, the elements of the array are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If omitted, the array is unpacked; that is, the elements are mapped in storage to optimize access time rather than to conserve space. (The array itself is always mapped into an addressable memory location.) For further information on how data is stored in memory, refer to appendix D, Data Representation in Memory.

If the array contains structured types (such as records), the elements of that type (the fields in the records) are not automatically packed. The structured type itself must be declared packed.

lower_bound

A constant integer expression that specifies the lower bound of the adaptable array. This parameter is optional, but its use is encouraged. Omission of this parameter (only the * appears) indicates it is an adaptable bound of type integer.

type

Type of the elements within the array. The type can be any defined type except an adaptable type (that is, an adaptable string, array, record, sequence, or heap). All elements must be of the same type.

Only one dimension can be adaptable in an array and that dimension must be the outermost (first one in the declaration).

Adaptable arrays adapt to a specific range of subscripts. An adaptable array can adapt to any array with the same packing attribute, equivalent subscript bounds, and equivalent component types. If a lower bound is specified in the adaptable array declaration, both arrays must also have the same lower bound.

Adaptable array types are equivalent if they have the same packing attributes and equivalent component types, and if their corresponding array and component subscript bounds are equivalent. Two subscript bounds that contain asterisks only are always equivalent. Two subscript bounds that contain identical lower bounds are equivalent.

Adaptable Records

An adaptable record contains zero or more fixed fields followed by one adaptable field that is a field of an adaptable type.

The format used for specifying an adaptable record is:

{*PACKED*} **RECORD**
  {*fixed_field_name* : {*ALIGNED* {[*offset MOD base*]}} *type*}...†
  adaptable_field_name : {*ALIGNED* {[*offset MOD base*]}}
    adaptable_type
RECEND

PACKED

Optional packing parameter. When specified, the fields of a record are mapped in storage in a manner that conserves storage space, possibly at the expense of access time. If omitted, the record is unpacked; that is, the fields are mapped in storage to optimize access time rather than to conserve space. For further information on how data is stored in memory, refer to appendix D, Data Representation in Memory.

If a field is a structured type (such as another record), the elements of that type are not packed automatically. The structured type itself must be declared packed.

fixed_field_name

Name identifying a particular fixed field. The name must be unique within the record.

ALIGNED

Optional alignment parameter. If specified, it can appear alone, or with an offset in the form:

ALIGNED [offset MOD base]

When a field is aligned, it is mapped in storage so that it is directly addressable. This means the field begins on an addressable boundary to facilitate rapid access to the field. This may negate some of the effect of packing the record. For further information, refer to Alignment earlier in this chapter.

† If more than one fixed (nonadaptable) field is specified, they must be separated by commas.

[offset MOD base]

Optional offset to be used in conjunction with the ALIGNED parameter. This offset causes the field to be mapped to a particular hardware address relative to the specified base and offset. Filler is created if necessary to ensure that the field begins on the specified addressable unit.

offset

An integer constant. Offset must be less than base.

base

An integer constant that must be divisible by 8. For automatic variables, the base can only be 8.

type

Any defined type, including another record, but not an adaptable type.

adaptable_field_name

Name identifying the adaptable field.

adaptable_type

An adaptable type.

An adaptable record can adapt to any record whose types are the same except for the last field. That last field must be one to which the adaptable field can adapt.

Two adaptable record types are equivalent if they have the same packing attributes, the same alignment, the same number of fields, and corresponding fields with identical names and equivalent types.

Adaptable Sequences

The format used for specifying an adaptable sequence is:

SEQ (*)

An adaptable sequence can adapt to a sequence of any size.

Two adaptable sequence types are always equivalent.

Adaptable Heaps

The format used for specifying an adaptable heap is:

HEAP (*)

An adaptable heap can adapt to a heap of any size.

Two adaptable heap types are always equivalent.

Expressions and Statements 5

This chapter describes expressions and statements that can be used within a CYBIL program, procedure, or function.

Expressions and Statements 5

Expressions

Expressions are made up of operands and operators. Operators act on operands to produce new values. (Constant expressions are evaluated to provide values for constants. Refer also to Constant Expressions in chapter 2.)

In general, operations involving nonequivalent types are not allowed; one type cannot be used where another type is expected. Exceptions are noted in the following descriptions.

Operands

Operands hold or represent the values to be used during evaluation of an expression. An operand can be a variable, constant, name of a constant, set value constructor, function reference (either standard function or user-defined function), pointer to a procedure name, pointer to a variable, or another expression enclosed in parentheses.

The value of a variable being used as an operand is the last value assigned to it. A constant name is replaced by the constant value associated with it in the constant declaration.

A function reference causes the function to be executed; the value returned by the function takes the place of the function reference in the expression.

Operators

Operators cause an action to be performed on one operand or a pair of operands. Many of the operators can be used only on basic types; they will be noted in their individual descriptions. Some operators can be used on sets. Although they are discussed in the individual descriptions that follow, there is also a separate description in this chapter on set operations.

An operation on a variable or component of a variable that has an undefined value will produce an undefined result.

There are five kinds of operators, many of which are identified by reserved symbols. They are listed here in the order in which they are evaluated from highest to lowest precedence.

- Negation operator (NOT)
- Multiplication operators (* , DIV, / , MOD, and AND)
- Sign operators (+ and -)
- Addition operators (+ , - , OR, and XOR)
- Relational operators (< , <= , > , >= , = , < > , and IN)

In the relational operators that consist of two symbols (that is, <=, >=, and < >), the symbols cannot be separated by a space or by any other character; they must appear together.

When an expression contains two or more operators of the same precedence, operations are performed from left to right. The only way to explicitly change the order of evaluation is to use parentheses. Parentheses indicate that the expression inside them should be evaluated first.

Negation Operator

The negation operator, NOT, applies only to boolean operands.

NOT TRUE equals FALSE. NOT FALSE equals TRUE.

Multiplication Operators

The multiplication operators perform multiplication and set intersection (*), integer quotient division (DIV), real quotient division (/), remainder division (MOD), and the logical AND operation (AND). Table 5-1 shows the multiplication operators, the permissible types of their operands, and the type of result they produce.

Table 5-1. Multiplication Operators

Operator	Operation	Type of Operands	Type of Result
*	Multiplication	Integer or subrange of integer	Integer
		Real	Real
*	Set intersection	Set of a scalar type	Set of the same type
DIV	Integer quotient†	Integer or subrange of integer	Integer
/	Real quotient	Real	Real
MOD	Remainder††	Integer or subrange of integer	Integer
AND	Logical AND†††	Boolean	Boolean

† Integer quotient refers to the whole number that results from a division operation. The remainder is ignored. A more formal definition is: for positive integers a, b, and n,

a DIV b = n

where n is the largest integer such that b * n <= a.

For one or two negative integers,

(-a) DIV b = (a) DIV (-b) = - (a DIV b) and
(-a) DIV (-b) = a DIV b

†† Remainder refers to the remainder of a division operation. A more formal definition is:

a MOD b = a - (a DIV b) * b

††† TRUE AND FALSE = FALSE
TRUE AND TRUE = TRUE
FALSE AND FALSE = FALSE
FALSE AND TRUE = FALSE

When the first operand is FALSE, the second operand is never evaluated.

Sign Operators

The sign operators perform the identity operation (+) and sign inversion and set complement operation (-). Table 5-2 shows the sign operators, the permissible types of their operands, and the type of result they produce.

Table 5-2. Sign Operators

Operator	Operation	Type of Operands	Type of Result
+	Identity (indicates a positive operand)	Integer	Integer
		Real	Real
-	Sign inversion (indicates a negative operand)	Integer	Integer
		Real	Real
-	Set complement	Set of a scalar type	Set of the same type

Addition Operators

The addition operators perform addition and set union (+), subtraction, boolean difference, and set difference (-), the logical OR operation (OR), and the exclusive OR operation (XOR). Table 5-3 shows the addition operators, the permissible types of their operands, and the type of result they produce.

Table 5-3. Addition Operators

Operator	Operation	Type of Operands	Type of Result
+	Addition	Integer of subrange of integer	Integer
+	Set union	Set of a scalar type	Set of the same type
-	Subtraction	Integer or subrange of integer	Integer
		Real	Real
-	Boolean difference†	Boolean	Boolean
-	Set difference	Set of a scalar type	Set of the same type
OR	Logical OR††	Boolean	Boolean
XOR	Exclusive OR†††	Boolean	Boolean
	Symmetric difference	Set of a scalar type	Set of the same type

† TRUE - TRUE = FALSE
TRUE - FALSE = TRUE
FALSE - TRUE = FALSE
FALSE - FALSE = FALSE

†† TRUE OR TRUE = TRUE
TRUE OR FALSE = TRUE
FALSE OR TRUE = TRUE
FALSE OR FALSE = FALSE

When the first operand is TRUE, the second operand is never evaluated.

††† TRUE XOR TRUE = FALSE
TRUE XOR FALSE = TRUE
FALSE XOR TRUE = TRUE
FALSE XOR FALSE = FALSE

Relational Operators

The relational operators (<, <=, >, >=, =, <>, and IN) test for the truth or falsity of these given conditions: less than (<), less than or equal to or subset of a set (<=), greater than (>), greater than or equal to or a superset of a set (>=), equal to or set identity (=), not equal to or set inequality (< >), and set membership (IN).

Because relational operators are valid on so many different types, some special points about each type are noted next. Following these comments, table 5-4 lists the relational operators and the permissible types of their operands; they always produce a boolean type result.

Comparison of Scalar Types

The comparison operators (< , <= , > , >= , = , and < >) are allowed only between operands of the same scalar type or between a substring of length 1 and a character.

For integer type operands, the relationships all have their usual meaning.

For character type operands, each character is essentially mapped to its corresponding integer value according to the ASCII collating sequence. (This is the same operation performed by the $INTEGER function described in chapter 6.) The operands and relational operators are then evaluated using the characters' integer values.

For boolean type operands, FALSE is always considered to be less than TRUE.

For ordinal type operands, operands are equal only if they are the same value; otherwise, they are not equal. For the other relational operators, each ordinal is essentially mapped to the corresponding integer value of its position in the ordinal list where it is defined. (This is the same operation performed by the $INTEGER function described in chapter 6.) The operands and relational operators are then evaluated using the ordinals' integer values. For an example, refer to the discussion of ordinal types under Scalar Types in chapter 4.

Operands that are a subrange of a scalar type can be compared with operands of the same type, including another subrange of the same type.

Comparison of Floating-Point Types

All of the comparison operators are valid between operands of the real type.

Comparison of Pointer Types

Two pointers can be compared if they are pointers to equivalent or potentially equivalent types. (For further information on equivalent types, refer to Equivalent Types in chapter 4.) For potentially equivalent types, one or both of the pointers can be pointers to adaptable or bound variant types. The current type of such a pointer must be equivalent to the type of the pointer with which it is being compared; if it is not, the operation is undefined.

Pointers can be compared for equality and inequality only. Two pointers are equal if they designate the same variable or if they both have the value NIL. A pointer of any type can be compared with the value NIL. Two pointers to a procedure are equal if they designate the same declaration of a procedure.

Comparison of Relative Pointers

Two relative pointers can be compared only if they are of equivalent types. Two relative pointers are equal if they can be converted to equal pointers using the #PTR function (described in chapter 6).

Comparison of String Types

All of the comparison operators are valid between operands that are strings. If the lengths of the two string operands are unequal, spaces are appended to the right of the shorter string to fill the field.

Strings are compared character by character from left to right; that is, each character from one string is compared with the character in the corresponding position of the second string. Each character is compared using the same method as for operands of character type; the integer value of the character, when mapped to the ASCII collating sequence, is used.

Comparison of Sets and Set Membership

Comparison operators have slightly different meanings for sets than for other types. The only comparison operators valid for sets are: = (meaning identical to), < > (meaning different from), <= (meaning the left operand is contained in the right operand), and >= (meaning the left operand contains the right operand). These operators are valid between two sets of the same type. Their exact meanings are detailed later in this chapter under Set Operators.

The other relational operator for sets is IN. A specified operand is IN a set if that operand is a member of the set. The set must be the same type or a subrange of the same type as the operand. The operand can be a subrange of the type of the set.

Comparison of Other Types

Invariant records can be compared for equality and inequality only. Two equivalent records are equal if their corresponding fields are equal.

The following types cannot be compared:

- Arrays or structures that contain an array as a component or field.
- Variant records.
- Sequences.
- Heaps.
- Records that contain a field of one of the preceding types.

However, pointers to these types can be compared.

Table 5-4. Relational Operators

Operator	Operation	Type of Left Operand	Type of Right Operand
<	Less than	Any scalar type	The same scalar type
<=	Less than or equal to	A string	A string of the same length
>	Greater than		
>=	Greater than or equal to	A string of length 1†	A character
=	Equal to		
< >	Not equal to	A character	A string of length 1†
IN	Set membership	Any scalar type	A set of the same type
		A string of length 1†	A set of character type
=	Equality (also called identity)	A set of any scalar type	A set of the same type
< >	Inequality		
<=	Is contained in		
<=	Contains		
= < >	Equality Inequality	A nonvariant record type containing no arrays	The same type
		Any pointer type or the value NIL	The same type or the value NIL

† The string of length 1 has the form
STRING(position)
where the length is implied. The form
STRING(position,1)
is not valid in this case.

Set Operators

The set operators have already been mentioned briefly in the preceding sections on multiplication, sign, addition, and relational operators. This section discusses all of them and details how they are used with sets.

The set operators perform assignment, union (+), intersection (*), difference (-), symmetric difference (XOR), negation (-), identity or equality (=), inequality (<>), inclusion (<=), containment (>=), and membership (IN).

Assignment is discussed under Sets in chapter 4. The next five operations (union, intersection, difference, symmetric difference, and negation) all produce results that are sets. They are described in table 5-5. The remaining operations (identity, inequality, inclusion, containment, and membership) produce boolean results. They are described in table 5-6.

The relational operations described in table 5-6 take place only after any operations described in table 5-5 have been performed.

Table 5-5. Operations That Produce Sets

Operator	Operation	Description of Operation
+	Union	The resulting set consists of all members of both sets. The result of A + B is all elements of sets A and B.
-	Difference	The resulting set consists of the members in the lefthand set that are not in the righthand set. The result of A - B is the elements of A that are not in B. This operation differs from negation in that two operands are present.
*	Intersection	The resulting set consists of the members that are in both sets. The result of A * B is all elements that are in both A and B.
-	Negation (complement)	The resulting set consists of the members of the set's type that are not in the set. The result of -A is all elements of A's type that are not in A. This operation differs from the difference operation in that only one operand is present.
XOR	Symmetric difference	The resulting set consists of the members of either but not both sets. The result of A XOR B is all elements in A or B that are not common to both A and B.

Table 5-6. Operations That Produce Boolean Results

Operator	Operation	Description of Operation
=	Equality (identity)	The resulting value is TRUE if every member of one set is present in the other set and vice versa. A = B is TRUE if every element of A is in B and every element of B is in A. It is also TRUE if A and B are both empty sets. In any other case, it is FALSE.
>	Inequality	The resulting value is TRUE if not every member of one set is a member of the other set. A < > B is TRUE if A = B is FALSE.
<=	Inclusion	The resulting value is TRUE if every member of the lefthand set is also a member of the righthand set. A <= B is TRUE if every element of A is in B. It is also TRUE if A is an empty set. In all other cases, it is FALSE.
>=	Containment	The resulting value is TRUE if every member of the righthand set is also a member of the lefthand set. A >= B is TRUE if every element of B is in A (that is, B <= A).
IN	Membership	This operation differs somewhat from the others in that it can specify as an operand a value or a variable rather than a set. It has the form scalar IN set where scalar can be a value (including a subrange) or a variable. The resulting value is TRUE if the scalar is of the same type as the type of the set, and is an element within the set. A IN B is TRUE if A is the same type as the set B and A is an element of B.

Statements

Statements indicate actions to be performed. Unlike declarations, statements can be executed. They can appear only in a program, procedure, or function.

A statement list is an ordered sequence of statements. In a statement list, a statement is separated from the one following it by a semicolon. Two consecutive semicolons indicate an empty statement, which means no action.

Statements can be divided into four types depending on their purpose or nature:

- Assignment
- Structured
- Control
- Storage management

Assignment Statement

The assignment statement assigns a value to a variable.

The format of the assignment statement is:

name := expression

name
Name of a variable previously declared.

expression
An expression that meets the requirements stated earlier in this chapter. Any constant or variable contained in the expression must be defined and have a value assigned.

This statement is similar to the initialization part of the VAR declaration where you can assign an initial value to a variable. (For further information on initialization, refer to Variable Declaration in chapter 3.) The assignment statement allows you to change that value at any point in the program. The expression is evaluated and the result becomes the current value of the named variable.

The variable cannot be:

- A read-only variable.
- A formal value parameter of the procedure that contains the assignment statement.
- A bound variant record.
- The tag field name of a bound variant record.
- A heap.
- An array or record that contains a heap.

The type of the expression must be equivalent to the type of the variable, with the exceptions discussed next. Both types can be subranges of equivalent types.

A character, string, or substring variable can be assigned the value of a character expression, a string, or a substring. If you assign a value that is shorter than the variable or substring to which it is being assigned, spaces are added to the right of the shorter string to fill the field. If you assign a value that is longer than the variable or substring, the value is truncated on the right. Assigning strings or substrings that overlap is not a valid operation, for example, STRING_1 := STRING_1(3,7); results are unpredictable.

If the variable is a pointer, its scope must be less than or equal to the scope of the data to which it is pointing. For example, a static pointer variable should not point to an automatic variable local to a procedure. When the procedure is left, the pointer variable will be pointing at undefined data.

A pointer to a bound variant record can be assigned a pointer to a variant record that is not bound and is otherwise equivalent.

An adaptable pointer can be assigned either a pointer to a type to which it can adapt, or an adaptable pointer than has been adapted to one of those types. Both the type of the expression and its value are assigned, thus setting the current type of the adaptable pointer.

Any fixed pointer except a pointer to sequence can be assigned a pointer to cell. After the assignment, the #LOC function (described in chapter 6) performed on the fixed pointer would return the same value as the pointer to cell.

A pointer to cell can be assigned any pointer type. The value assigned is a pointer to the first cell allocated for the variable to which the pointer being assigned points.

When assigning pointers, remember that generally the object of a pointer has a different lifetime than the pointer variable. Automatic variables are released when the block in which they are declared has been executed. Allocated variables no longer exist when they are explicitly released with the FREE statement. An attempt to reference a variable beyond its lifetime causes an error and unpredictable results to occur.

A variant record can be assigned a bound variant record of types that are otherwise equivalent.

The colon (:) and equals sign (=) together are called the assignment operator. When used as the assignment operator, there can be no spaces or comments between the two symbols.

Structured Statements

A structured statement is one that actually contains one or more statements. The statements contained in a structured statement are called, collectively, a statement list. The structured statement determines when the statement list contained in it will be executed.

There are four structured statements:

BEGIN	Provides a logical grouping of statements that performs a specific function.
FOR	Executes a list of statements while a variable is incremented or decremented from an initial value to a final value.
REPEAT	Executes a list of statements until a specified condition is true. The test is made after each execution of the statements.
WHILE	Executes a list of statements while a specified condition is true. The test is made before each execution of the statements.

BEGIN Statement

The BEGIN statement executes a single statement list once; there is no repetition. This statement provides for a logical grouping of statements that performs a particular function and can improve readability.

The format of the BEGIN statement is:

{*/label/*}
BEGIN
   statement list;
END {*/label/*}**;**

label

Name that identifies the BEGIN statement and the statement list within it. Use of labels is optional. If a label is used before BEGIN, it is not required after END but is encouraged. If labels are used in both places, they must match. The label name must be unique within the block in which it is used.

statement list

One or more statements.

Declarations are not allowed with the BEGIN statement. Execution of the BEGIN statement ends when either the last statement in the list is executed or control is explicitly transferred from within the list.

FOR Statement

The FOR statement executes a statement list repeatedly while a special variable ranges from an initial value to a final value. There are two formats for the FOR statement: one that increments the variable and one that decrements the variable.

The format that increments the variable is:

{/*label*/}
FOR name := initial_value TO final_value DO
  statement list;
FOREND {/*label*/}**;**

The format that decrements the variable is:

{/*label*/}
FOR name := initial_value DOWNTO final_value DO
  statement list;
FOREND {/*label*/}**;**

label

Name that identifies the FOR statement and the statement list in it. Use of labels is optional. If a label is used before FOR, it is not required after FOREND but is encouraged. If labels are used in both places, they must match. The label name must be unique within the block in which it is used.

name

Name of the variable that controls the number of repetitions of the statement list. It keeps track of the number of iterations performed or the current position within the range of values.

initial_value

Scalar expression specifying the initial value assigned to the variable.

final_value

Scalar expression specifying the final value to be assigned to the variable if the statement ends normally. If the statement ends abnormally or as the result of an EXIT statement, this may not be the actual final value.

statement list

One or more statements.

The variable, initial value, and final value must be of equivalent scalar types or subranges of equivalent types. The variable cannot be assigned a value within the statement list, or be passed as a reference parameter to a procedure called within the statement list. Either condition causes a fatal compilation error. The variable cannot be an unaligned component of a packed structure.

When CYBIL encounters a FOR statement that increments (one containing the TO clause), it evaluates the initial value and final value. If the initial value is greater than the final value, the FOR statement ends and execution continues with the statement following FOREND; the statement list is not executed. If the initial value is less than or equal to the final value, the initial value is assigned to the control variable and the statement list is executed. Then, the control variable is incremented by one value and, for each increment, the statement list is executed. This sequence of actions continues through the final value. For example, the statement

```
FOR i = 1 TO 5 DO
    .
    .
    .
FOREND;
```

causes the statement list to be executed five times, that is, while I takes on values from 1 through 5. Then the FOR statement ends and execution continues with the statement following FOREND.

When CYBIL encounters a FOR statement that decrements (one containing the DOWNTO clause), it performs essentially the same process. If the initial value is less than the final value, the FOR statement ends and execution continues with the statement following FOREND. If the initial value is greater than or equal to the final value, the initial value is assigned to the control variable and the statement list is executed. The control variable is decremented by one value and, for each decrement, the statement list is executed. When the control variable reaches the final value and the statement list is executed the last time, the FOR statement ends.

The initial value and final value expressions are evaluated once, when the statement is entered; the values are then held in temporary locations. Thus, subsequent assignments to initial value and final value have no effect on the execution of the FOR statement.

When a FOR statement completes normally, the value of the control variable is that of the final value specified in the statement. This may not be the case if the statement ends abnormally or ends as a result of an EXIT statement.

Example:

Integer values are often used in FOR statements, but any scalar type can be used. The following example executes a statement list while the value of a character variable is incremented.

```
FOR control := 'a' TO 'z' DO
    .
    .
    .
FOREND;
```

Each time the statement list is performed, the value of CONTROL increases by one value, following the normal sequence of alphabetic characters from A through Z; that is, after the statement list is executed once, the value of CONTROL changes to B, and so on until the list has been executed 26 times.

REPEAT Statement

The REPEAT statement executes a statement list repeatedly until a specific condition is true.

The format of the REPEAT statement is:

{/*label*/}
REPEAT
  statement list;
UNTIL expression;

label

Name that identifies the REPEAT statement and the statement list in it. Use of the label before REPEAT is optional; a label is not permitted after UNTIL. The label name must be unique within the block in which it is used.

statement list

One or more statements.

expression

A boolean type expression.

The statement list is always executed at least once. After the last statement in the list, the expression is evaluated. Every time the expression is FALSE, the statement list is executed again. When the expression is TRUE, the REPEAT statement ends and execution continues with the statement following the UNTIL clause.

The statement list can contain nested REPEAT statements.

Example:

In this example, the statement list (mod operation and assignments) is executed once. If J is not equal to zero, it is executed again and continues until J is equal to zero.

```
REPEAT
  k := i MOD j;
  i := j;
  j := k;
UNTIL j = 0;
```

WHILE Statement

The WHILE statement executes a statement list repeatedly while a specific condition is true.

The format of the WHILE statement is:

{/*label*/}
WHILE expression DO
  statement list;
WHILEND {/*label*/};

label

Name that identifies the WHILE statement and the statement list in it. Use of labels is optional. If a label is used before WHILE, it is not required after WHILEND but is encouraged. If labels are used in both places, they must match. The label name must be unique within the block in which it is used.

expression

A boolean type expression.

statement list

One or more statements.

If the boolean expression is evaluated as TRUE, the statement list is executed. After the last statement in the list, the expression is again evaluated. Every time the expression is TRUE, the statement list is executed. When the expression is FALSE, the WHILE statement ends and execution continues with the statement following WHILEND. If the expression is FALSE in the initial evaluation, the statement list is never executed.

Example:

In this example, the expression TABLE[I] < > 0 is evaluated; an element of the array TABLE is compared to zero. While the expression is true (the element is not zero), I is incremented. This causes the next element of the array to be checked. When the expression is false, the statement list is not executed. Execution continues with the statement following WHILEND. I is the position of an element in the array that is zero.

```
/check_for_zero/
  WHILE table[i] < > 0 DO
    i := i + 1;
  WHILEND /check_for_zero/;
```

The preceding example assumes, of course, that the array contains an element with the value zero. If not, the WHILE statement list executes in an infinite loop. In either the WHILE expression or the statement list, there must be a check. One solution is to set a variable, TABLE_MAX, to the maximum number of elements in the array and check it before executing the statement list, as in:

```
WHILE (i < table_max) AND (table[i] < > 0) DO
```

Now both expressions must be true before the statement list is executed. If either is false, execution continues following WHILEND.

Control Statements

A control statement can change the flow of execution of a program by transferring control from one place in the program to another.

There are five control statements:

IF	Executes one statement list if a given condition is true; ends the statement or executes another statement list if the condition is false.
CASE	Executes one statement list out of a set of statement lists depending on the value of a given expression.
CYCLE	Causes the remaining statements in a repetitive statement (FOR, REPEAT, or WHILE) to be skipped and the next iteration of the statement to take place.
EXIT	Unconditionally stops execution within a procedure, function, or a structured statement (BEGIN, REPEAT, WHILE, and FOR).
RETURN	Returns control from a procedure or function to the point at which it was called.

Procedure and function calls also transfer control of an executing program. Functions are discussed in chapter 6 and procedures are discussed in chapter 7.

IF Statement

The IF statement executes or skips a statement list depending on whether a given condition is true or false.

The format of the IF statement is:

IF expression THEN
  statement list;
{*ELSEIF expression THEN*
  statement list;}...
{*ELSE*
  statement list;}
IFEND;

expression

A boolean expression.

statement list

One or more statements.

The ELSEIF and ELSE clauses are optional. The ELSEIF clause contains another test condition that is evaluated only if the preceding condition (expression) is false. The ELSE clause provides a statement list that is executed unconditionally when the preceding expression is false.

When an expression is evaluated as true, the statement list following the reserved word THEN is executed. When the list is completed, execution continues with the first statement following IFEND. If the expression is false, execution continues with the next clause or reserved word in the IF statement format (that is, ELSEIF, ELSE, or IFEND).

If the next reserved word in the IF statement format is IFEND, execution continues with the first statement following it.

If the next reserved word is ELSEIF, the expression contained in that clause is evaluated; if true, the statement list that follows is executed. Otherwise, execution continues with the next reserved word in the IF statement format.

If the next reserved word is ELSE, the statement list that follows is always executed. You get to this point only if the preceding expression(s) is false.

Additional IF statements can be contained (nested) in any of the statement lists. A consistent style of indentation or spacing greatly improves readability of such statements.

If the ELSE clause is included in a nested IF statement, the clause applies to the most recent IF statement.

Examples:

In this example, Y is assigned to X if and only if X is less than Y.

```
IF x < y THEN
  x := y;
IFEND;
```

In the next example, Z is always assigned one of the values 1, 2, 3, or 4 depending on the value of X.

```
IF x <= 5 THEN
  z := 1;
ELSEIF x > 30 THEN
  z := 2;
ELSEIF x = 15 THEN
  z := 3;
ELSE
  z := 4;
IFEND;
```

CASE Statement

The CASE statement executes one statement list out of a set of lists based on the value of a given expression.

The format of the CASE statement is:

CASE expression OF
= value {,*value*}... **=**
  statement list;
{= *value* {,*value*}... =
  statement list;}...
{*ELSE statement list;*}
CASEND;

expression

A scalar expression. The expression must be of the same type as the value or values that follow.

value

One or more constant scalar expressions or a subrange of constant scalar expressions. A subrange indicates that all of the values included in the subrange are acceptable values. If two or more values are specified, they are separated by commas. The values must be of the same type as the expression. Values can be in any order, not strictly sequential. Values must be unique within the CASE statement.

statement list

One or more statements.

You define a set of possible values that a variable or expression can have. With one or more of the values you associate a statement list using the format:

```
= value =
   statement list;
```

When the CASE statement is executed, the expression is evaluated and the statement list associated with the current value of the expression is executed. If the current value is not found among those in the CASE statement, execution continues with the ELSE clause. If ELSE is omitted and the value is not found in the CASE statement, an error occurs at execution time. After any one of the statement lists is executed, execution continues with the statement following CASEND.

Examples:

In this example, I is a variable that is expected to take on one of the values 1 through 4. If its value is 1, the first statement list (X := X + 1) is executed and control goes to the statement following CASEND. If the value of I is 2, the second list is executed, and so on.

```
CASE i OF
  = 1 =
    x := x + 1;
  = 2 =
    x := x + 2;
  = 3 =
    x := x + 3;
  = 4 =
    x := x + 4;
CASEND;
```

In the next example, OPERATOR is a variable that is expected to take on values of PLUS, MINUS, or TIMES. Depending on the current value of OPERATOR, the associated statement is executed.

```
CASE operator OF
  = plus =
    x := x + y;
  = minus =
    x := x - y;
  = times =
    x := x * y;
CASEND;
```

CYCLE Statement

The CYCLE statement can be included in the statement list of a repetitive statement (FOR, REPEAT, or WHILE) and causes any statements following it to be skipped and the next iteration of the repetitive statement to take place.

The format of the CYCLE statement is:

CYCLE /label/

label

Name that identifies the repetitive statement in which the CYCLE statement is contained.

The CYCLE statement is usually used in conjunction with an IF statement, as in:

```
/label/
repetitive statement
   IF expression THEN
   CYCLE /label/;
   IFEND;
   remainder of statement list;
end of repetitive statement;
```

The IF statement tests for a condition that, if true, causes the CYCLE statement to be executed. Then the remaining statements of the repetitive statement are skipped and execution continues with whatever would normally follow the statement list, either another cycle of the repetitive statement or the next statement following the end of the repetitive statement. If the condition in the IF statement is false, the remaining statements in the repetitive statement are executed.

If not contained in a repetitive statement, the CYCLE statement is diagnosed as a compilation error.

Example:

This example finds the smallest element of an array TABLE. On the first execution, X (the first element of the array) is assumed to be smallest. If X is smaller than succeeding elements of the array, the CYCLE statement is executed; the remainder of the statements are then skipped, and the next iteration of the FOR statement occurs. If an element smaller than X is found, the CYCLE statement is ignored and the rest of the statement list is processed; X is replaced by the smaller element. If N has not yet been reached, the FOR statement continues. When N is reached, X will contain the smallest element of the array.

```
x := table[1];

/find_smallest/
 FOR k := 2 TO n DO
   IF x < table[k] THEN
     CYCLE /find_smallest/;
   IFEND;
   x := table[k];
 FOREND /find_smallest/;
```

EXIT Statement

The EXIT statement causes an unconditional exit from a procedure, function, or a structured statement (BEGIN, FOR, REPEAT, and WHILE).

The format of the EXIT statement is:

EXIT name;

name

Name that identifies the procedure, function, or statement. For a procedure or function, it is the procedure or function name. For a structured statement, it is the statement label; in this case the format could be shown as EXIT /label/.

When the EXIT statement is encountered, execution of the named procedure, function, or statement is automatically stopped and execution resumes with the statement that would follow normal completion. For a procedure or function, it is the statement that would normally follow the procedure or function call. For a structured statement, it is the statement following the end of the structured statement (END, FOREND, UNTIL expression, and WHILEND).

The EXIT statement must be within the scope of the procedure, function, or statement it names. Otherwise, it has no meaning and is diagnosed as a programming error.

With a single EXIT statement, you can exit several levels of procedures, functions, or statements; they need not be exited separately. If the EXIT statement is executed in a nested recursive procedure or function, it is the most recent invocation of the procedure or function and any intervening procedures or functions that are exited.

RETURN Statement

The RETURN statement completes the execution of a procedure or function and returns control to the program, procedure, or function that called it.

The format of the RETURN statement is:

RETURN;

If omitted at the end of a procedure or function, the RETURN statement is assumed.

Storage Management Statements

Storage management statements allow you to manipulate components of sequence and heap types, and put variables in the run-time stack.

There are five storage management statements:

RESET	Resets the pointer in a sequence or releases all the variables in a user-defined heap.
NEXT	Creates or accesses the next element of a sequence given a starting element.
ALLOCATE	Allocates storage for a variable in a heap.
FREE	Releases a variable from a heap.
PUSH	Allocates storage for a variable in the run-time stack.

Sequences use the RESET and NEXT statements. Heaps use the RESET, ALLOCATE, and FREE statements. The run-time stack uses the PUSH statement. (Refer to Storage Types in chapter 4 for further information on sequences and heaps.) The NEXT and ALLOCATE statements can also be used to allocate space for a segment access file. Accessing a file as a memory segment is described in the CYBIL File Interface manual. That manual also compares use of the default heap and run-time stack with use of a segment access file for data storage.

In the NEXT, ALLOCATE, and PUSH statements, you must specify a pointer to the variable to be manipulated so that sufficient space can be allocated for that type. This pointer can be a pointer to a fixed type, a pointer to an adaptable type, or a pointer to a bound variant record type. Space is then allocated for a variable of the type to which the pointer can point. This pointer is also used to access the variable. When space is allocated, CYBIL returns the address of the variable to the pointer. Therefore, to reference a variable in a sequence, heap, or the run-time stack, you indicate the object of the pointer in the form: pointer name ^.

If a fixed type pointer is specified, the statement uses a variable of the type designated by that pointer variable. If an adaptable type pointer or bound variant record type pointer is specified, you must also indicate the size of the adaptable type or the tag field of the variant record to be used. This causes a fixed type to be set and the adaptable or bound variant record pointer designates a variable of that fixed type. That particular fixed type is designated until it is reset by a subsequent assignment or another storage management statement.

To indicate the size of an adaptable pointer or the tag field of a bound variant record pointer, you use the format:

pointer : [size]

pointer

Name of an adaptable pointer variable or a bound variant record pointer variable.

size

Fixed amount of space required for the variable designated by pointer. You set the size of the adaptable type the same way you specify the size of the corresponding unadaptable (fixed) type. For example, in a variable or type declaration, you specify the size of a fixed array with subscript bounds, usually a subrange of "scalar expression..scalar expression". You set the size of an adaptable array here using the same form. The forms used to set the size of all possible adaptable types are summarized as follows. For more detailed information, refer to the descriptions of the corresponding fixed types in chapter 4.

Pointer Type	**Form Used to Set Size**
Adaptable array	scalar expression .. scalar expression
Adaptable string	A positive integer expression specifying the length of the string
Adaptable heap	[{*REP positive integer expression OF*} fixed type name {,{*REP positive integer expression OF*} *fixed type name*}...]
Adaptable sequence	[{*REP positive integer expression OF*} fixed type name {,{*REP positive integer expression OF*} *fixed type name*}...]
Adaptable record	One of the forms used for an adaptable array, string, heap, or sequence
Bound variant record	A scalar expression or one or more constant scalar expressions followed by an optional scalar expression

If an adaptable array had a lower bound specified in its original declaration, the lower bound specified here must match that value. For an adaptable record, the form used must be a value and type to which the record can adapt. For a bound variant record, the order, types, and values used must be valid for a variant of the record; all but the last of the expressions must be constant expressions.

Examples:

This example declares a type that is an adaptable array named ADAPT_ARRAY. PTR is a pointer to that type. BUNCH is a heap with space for 100 integers. The heap BUNCH is reset; that is, any existing elements are released. Space is then allocated in the heap for a variable of the type designated by PTR. That variable is of type ADAPT_ARRAY (an array of integers) and it has fixed subscript bounds of from 1 through 15. PTR now points to that array.

```
TYPE
  adapt_array = array [1..*] of integer;
VAR
  ptr : ^ adapt_array,
  bunch : heap (rep 100 of integer);

RESET bunch;
ALLOCATE ptr : [1..15] IN bunch;
```

The following example shows the setting of an adaptable sequence. Notice that two sets of brackets are required in the PUSH statement.

```
VAR
  ptr : ^ SEQ (*);
PUSH ptr : [[REP 10 OF integer, REP 22 OF char]];
```

RESET Statement

The RESET statement operates on both sequences and heaps. In a sequence, it resets the pointer to the beginning of the sequence or to a specific variable within the sequence. In a heap, it releases all the variables in the heap.

The RESET statement must appear before the first NEXT statement (for a sequence) or ALLOCATE statement (for a user-defined heap). This ensures that the sequence is at the beginning or the heap is empty. If space is reserved (by a NEXT or ALLOCATE statement) before the RESET statement, the program is in error.

RESET in a Sequence

This statement sets the current element being pointed to in a sequence.

The format of the RESET statement in a sequence is:

RESET sequence_pointer { *TO variable_pointer* }

sequence_pointer
Name of a pointer to a sequence. This specifies the particular sequence.

variable_pointer
Name of a pointer to a particular variable within the sequence. If omitted, the pointer points to the first element of the sequence.

The value of the pointer variable must have been set with a NEXT statement for the same sequence or an error occurs. An error also occurs if the value of the pointer variable is NIL.

The RESET statement must appear before the first occurrence of a NEXT statement to reset the sequence to its beginning; otherwise, the program is in error.

RESET in a Heap

This statement releases the variables currently in a heap.

The format of the RESET statement in a heap is:

RESET heap

heap

Name of a heap type variable.

Space for the variables is released and their values become undefined.

The RESET statement must appear before the first occurrence of an ALLOCATE statement for a user-defined heap to ensure that the heap is empty; otherwise, the program is in error.

NEXT Statement

The NEXT statement sets the specified pointer to designate the current element of the sequence and then makes the next element in the sequence the current element. This essentially moves the pointer along the sequence allowing you to assign values to and access elements.

The format of the NEXT statement is:

NEXT pointer { : *[size]* } **IN sequence_pointer**

pointer
Name of a pointer to a fixed type, pointer to an adaptable type, or pointer to a bound variant record type. The type pointed to by the pointer is the type of the variable in the sequence. These pointers are described in detail under Storage Management Statements earlier in this section.

size
Size of an adaptable type or tag field of a bound variant record type. If omitted, the pointer must be a pointer to a fixed type. The forms used to specify size are described in detail under Storage Management Statements earlier in this section.

sequence_pointer
Name of a pointer to a sequence. This specifies the particular sequence.

After a RESET statement, the current element is always the first element of the sequence. A NEXT statement assigns to the specified pointer the address of the current (first) element, and then makes the next element (the second) the new current element. Thus, the order of variables in a sequence is determined by the order in which the NEXT statements are executed.

If the NEXT statement causes the new element to be outside the bounds of the sequence, the pointer is set to NIL. Before attempting to reference an element in a sequence, check for a NIL pointer value. Using a pointer variable with a value of NIL to access an element causes an error to occur.

The type of the pointer specified when data is retrieved from the sequence must be equivalent to the type of the pointer used when the same data was stored in the sequence; otherwise, the program is in error.

ALLOCATE Statement

The ALLOCATE statement allocates storage space for a variable of the specified type in the specified heap and then sets the pointer to point to that variable.

The format of the ALLOCATE statement is:

ALLOCATE pointer { *: [size]* } { *IN heap* }

pointer

Name of a pointer to a fixed type, adaptable type, or bound variant record type. These pointers are described in detail under Storage Management Statements earlier in this section.

size

Size of an adaptable type or tag field of a bound variant record type. If omitted, the pointer must be a pointer to a fixed type. The forms used to specify size are described in detail under Storage Management Statements earlier in this section.

heap

Name of a heap type variable. If omitted, the default heap is assumed.

If there is not enough space for the variable to be allocated, the pointer is set to NIL. Before attempting to reference a variable in a heap, check for a NIL pointer value. Using a pointer variable with a value of NIL to access data causes an error to occur.

The RESET statement must appear before the first occurrence of an ALLOCATE statement for a user-defined heap to ensure that the heap is empty; otherwise, the program is in error. (This is not allowed for the default heap.)

The lifetime of a variable that is allocated using the storage management statements is the time between the allocation of storage (with the ALLOCATE statement) and the release of storage (with the FREE statement). A variable allocated using an automatic pointer must be explicitly freed (using the FREE statement) before the block is left, or the space will not be released by the program. When the block is left, the pointer no longer exists and, therefore, the variable cannot be referenced. If the block is entered again, the previous pointer and the variable referenced by the pointer cannot be reclaimed.

FREE Statement

The FREE statement releases the specified variable from the specified heap.

The format of the FREE statement is:

FREE pointer { *IN heap* }

pointer
Name of the pointer variable that designates the variable to be released.

heap
Name of a heap type variable. If omitted, the default heap is assumed.

The variable's space in the heap is released and its value becomes undefined. The pointer variable designating the released variable is set to NIL. If the specified variable is not currently allocated in the heap, the effect is undefined.

Using a pointer variable with the value NIL to access data causes an error to occur. Releasing the NIL pointer is also an error.

PUSH Statement

The PUSH statement allocates storage space on the run-time stack for a variable of the specified type and then sets the pointer to point to that variable.

The format of the PUSH statement is:

PUSH pointer { *: [size]* }

pointer

Name of a pointer to a fixed type, adaptable type, or bound variant record type. These pointers are described in detail under Storage Management Statements earlier in this section.

size

Size of an adaptable type or tag field of a bound variant record type. If omitted, the pointer must be a pointer to a fixed type. The forms used to specify size are described in detail under Storage Management Statements earlier in this section.

If there is not enough space for the variable to be allocated, the pointer is set to NIL. The value of the variable that has just been allocated is undefined until a subsequent assignment to the variable is made.

You cannot release space on the run-time stack explicitly. It is released automatically when the procedure containing the PUSH statement is completed and control leaves the procedure. At that time, space for the variable is released and its value becomes undefined.

Example:

This example shows the declaration of a pointer variable named ARRAY_PTR that points to an adaptable array. The PUSH statement allocates space in the run-time stack for a fixed array of from 1 through 20 elements. Elements of the array can be referenced by ARRAY_PTR^[i], where i is an integer from 1 through 20.

```
VAR array_ptr : ^array [1..*] of integer;

PUSH array_ptr : [1..20];
```

Functions 6

[his chapter describes the functions that are predefined in CYBIL and lescribes how to define your own functions.

Functions 6

A function is one or more statements that perform a specific action and can be called by name from a statement elsewhere in a program. A reference to a function causes actual parameters in the calling statement to be substituted for the formal parameters in the function declaration and then the function's statements to be executed. Usually the function computes a value and returns it to the portion of the program that called it.

A function differs from a procedure in that the value returned for a function replaces the actual function reference within the statement. A function is a valid operand in an expression; the value returned by the function replaces the reference and becomes the operand.

The value of a function is the last value assigned to it before the function returns to the point where it was called. The reason for its return doesn't matter; it could complete normally or abnormally. If the function returns for any reason before a value is assigned to the function name, results are undefined.

Functions can be recursive; that is, a function can call itself. In that case, however, there must be some provision for ending the calls.

You can call functions that are already defined in the language or you can define your own functions. This chapter describes both.

Standard Functions

The functions described here are standard CYBIL functions. They can be used safely in variations of CYBIL available on other operating systems. The next section in this chapter, System-Dependent Functions, describes functions unique to CYBIL on NOS/VE.

The functions are described in alphabetical order according to the first alphabetic character.

$CHAR Function

The $CHAR function returns the character whose ordinal number within the ASCII collating sequence is that of a given expression.

The format of the $CHAR function call is:

$CHAR(expression)

expression

An integer expression whose value can be from 0 through 255.

If the value of the integer expression is less than 0 or greater than 255, an error occurs.

$INTEGER Function

The $INTEGER function returns the integer value of a given expression.

The format of the $INTEGER function call is:

$INTEGER(expression)

expression

An expression of type integer, subrange of integer, boolean, character, ordinal, or real.

If the expression is an integer expression, the value of that expression is returned.

If the expression is a boolean expression, 0 (zero) is returned for a false expression and 1 is returned for a true expression.

If the expression is a character expression, the ordinal number of the character in the ASCII collating sequence is returned.

If the expression is an ordinal expression, the ordinal number associated with that ordinal value is returned. The value returned for the first element of an ordinal type is zero, the second element is one, and so on.

If the expression is a real expression, the value of the expression is truncated to a whole number. If the number is in the range defined for integers, that number is returned; otherwise, an out-of-range error occurs.

#LOC Function

The #LOC function returns a pointer to the first cell allocated for a given variable.

The format of the #LOC function call is:

#LOC(name)

name

Name of a variable.

LOWERBOUND Function

The LOWERBOUND function returns the lower bound of an array's subscript bounds.

The format of the LOWERBOUND function call is:

LOWERBOUND(array)

array
An array variable or the name of a fixed array type.

The type of the value returned is same as the type of the array's subscript bounds.

Example:

Assuming the following declaration has been made

```
VAR
  x : array [1..100] of boolean,
  y : array ['a'..'t'] of integer;
```

the value of LOWERBOUND(X) is 1; the value of LOWERBOUND(Y) is 'A'.

LOWERVALUE Function

The LOWERVALUE function returns the smallest possible value that a given variable or type can have.

The format of the LOWERVALUE function call is:

LOWERVALUE(name)

name

A scalar variable or name of a scalar type.

The type of the value returned is the same as the given type.

Examples:

Assuming the following declaration has been made

```
VAR
  dozen : 1..12;
```

the value of LOWERVALUE(DOZEN) is 1.

After the declarations

```
TYPE
  t = (first, second, third);

VAR
  v : t;
```

the value of LOWERVALUE(V) is FIRST and the value of LOWERVALUE(T) is FIRST.

PRED Function

The PRED function returns the predecessor of a given expression.

The format of the PRED function call is:

PRED(expression)

expression
A scalar expression.

If the predecessor of the expression does not exist, the program is in error.

Example:

The following example declares two variables, WARM and COLD, each of which can take on ordinal values of the type SEASONS. The variable WARM is assigned the value SPRING while the variable COLD is assigned the value WINTER.

```
TYPE
  seasons = (winter, spring, summer, fall);

VAR
  warm : seasons,
  cold : seasons;

warm := spring;
cold := PRED(warm);
```

#PTR Function

The #PTR function returns a pointer that can be used to access the object of a relative pointer.

The format of the #PTR function call is:

#PTR(pointer_name {,*parent_name*}**)**

pointer_name
Name of the relative pointer variable.

parent_name
Name of the variable that contains the components being designated by relative pointers. If omitted, the default heap is used. The variable can be a string, array, record, heap, or sequence type (either fixed or adaptable).

Relative pointers cannot be used to access data directly. The #PTR function converts a relative pointer to a pointer in order to reference the object of the relative pointer.

The type of the object pointed to by the returned pointer is the same as the type of the object pointed to by the relative pointer. If the type of the parent variable associated with the specified relative pointer is not equivalent to the type of the specified parent variable, an error occurs.

For further information on relative pointers, refer to Pointer Types in chapter 4.

$REAL Function

The $REAL function returns the real number equivalent of a given integer expression.

The format of the $REAL function call is:

$REAL(expression)

expression

An integer expression.

#REL Function

The #REL function returns a relative pointer.

The format of the #REL function call is:

#REL(pointer_name {,*parent_name*})

pointer_name
Name of the direct pointer variable.

parent_name
Name of the variable that contains the components being designated by relative pointers. If omitted, the default heap is used. The variable can be a string, array, record, heap, or sequence type (either fixed or adaptable).

The type of the relative pointer's object is the same as the type of the given direct pointer's object. (This type was specified in the VAR declaration of the relative pointer variable.) The parent type of the relative pointer's object is the same as the type of the specified parent variable.

If the pointer specified in the function call does not designate an element of the parent variable, the result is undefined.

Relative pointer values can be generated solely through this function. For further information on relative pointers, refer to Pointer Types in chapter 4.

#SIZE Function

The #SIZE function returns the number of cells required to contain a given variable or a variable of a specified type.

The format of the #SIZE function call is:

#SIZE(name)

name

Name of a variable, fixed record type, or bound variant record type.

If the name of a bound variant record type is specified, the variant that requires the largest size is used.

STRLENGTH Function

The STRLENGTH function returns the length of a given string.

The format of the STRLENGTH function call is:

STRLENGTH(string)

string
A string variable, name of a string type, or adaptable string reference.

For a fixed string, the allocated length is returned as an integer subrange. For an adaptable string, the current length is returned.

SUCC Function

The SUCC function returns the successor of a given expression.

The format of the SUCC function call is:

SUCC(expression)

expression
A scalar expression.

If the successor of the expression does not exist, the program is in error.

Example:

The following example declares two variables, HOT and COOL, each of which can take on ordinal values of the type SEASONS. The variable HOT is assigned the value SUMMER while the variable COOL is assigned the value FALL.

```
TYPE
  seasons = (winter, spring, summer, fall);

VAR
  hot : seasons,
  cool : seasons;

hot := summer;
cool := SUCC(hot);
```

UPPERBOUND Function

The UPPERBOUND function returns the upper bound of an array's subscript bounds.

The format of the UPPERBOUND function call is:

UPPERBOUND(array)

array
An array variable or the name of a fixed array type.

The type of the value returned is the same as the type of the array's subscript bounds.

Examples:

Assuming the following declaration has been made

```
VAR
  x : array [1..100] of boolean,
  y : array ['a'..'t'] of integer;
```

the value of UPPERBOUND(X) is 100; the value of UPPERBOUND(Y) is 'T'.

In the following example, the value of UPPERBOUND(TABLE) is 50.

```
VAR
  table : ^ array [1..*] of cell;

allocate table : [1..50];
```

UPPERVALUE Function

The UPPERVALUE function returns the largest possible value that a given variable or type can have.

The format of the UPPERVALUE function call is:

UPPERVALUE(name)

name

A scalar variable or name of a scalar type.

The type of the value returned is the same as the given type.

Examples:

Assuming the following declaration has been made

```
VAR
  dozen : 1..12;
```

the value of UPPERVALUE(DOZEN) is 12.

After the declarations

```
TYPE
  t = (first, second, third);

VAR
  v : t;
```

the value of UPPERVALUE(V) is THIRD and the value of UPPERVALUE(T) is THIRD.

System-Dependent Functions

The functions described here can be used with CYBIL only on NOS/VE. Keep in mind that programs using these functions cannot be transported to other operating systems and run on variations of CYBIL.

To use these functions properly and efficiently, you should be familiar with basic hardware concepts of your computer system. This information can be found in volume II of the virtual state hardware reference manual.

The functions are described in alphabetic order according to the first alphabetic character.

#ADDRESS Function

The #ADDRESS function accepts a ring number, segment number, and byte offset and returns a value that is of type pointer to cell.

The format of the #ADDRESS function call is:

#ADDRESS(ring, segment, offset)

ring
Ring number, ranging from 1 through 15.

segment
Segment number, ranging from 0 through 4,095.

offset
Byte offset, ranging from -80000000 hexadecimal through 7FFFFFFF hexadecimal.

#FREE_RUNNING_CLOCK Function

The #FREE_RUNNING_CLOCK function returns the value of the free running microsecond clock.

The format of the #FREE_RUNNING_CLOCK function call is:

#FREE_RUNNING_CLOCK(port)

port

An integer expression whose value is 0 or 1. It specifies the memory port to be used for reading the clock.

The integer value returned is that of the free running clock that is maintained within the memory connected to the specified processor memory port.

For further information on the free running microsecond clock and memory ports, refer to volume II of the virtual state hardware reference manual.

#OFFSET Function

The #OFFSET function accepts a direct pointer and returns the integer value of the signed offset (byte number) contained in the pointer.

The format of the #OFFSET function call is:

#OFFSET(pointer)

pointer

Name of a direct pointer expression.

A pointer consists in part of the process virtual address (PVA) of the first byte of the object to which it is pointing. An element of the PVA is the byte number. This byte number is the signed offset returned.

For further information on PVAs, refer to volume II of the virtual state hardware reference manual.

#PREVIOUS_SAVE_AREA Function

The #PREVIOUS_SAVE_AREA function returns a pointer to the first cell of the previous save area.

The format of the #PREVIOUS_SAVE_AREA function call is:

#PREVIOUS_SAVE_AREA

A procedure uses an area called a stack frame to store its dynamic variables. If another procedure is called, hardware saves certain registers of the calling procedure and puts them in a stack frame save area. These registers contain the information required for the calling procedure to resume normal execution when control is returned by the called procedure.

If procedure calls are nested, each subsequent call creates its own stack frame save area and the last save area becomes the "previous save area." Pointers are kept to link the previous save areas so that as procedures complete and return, the system works back through the previous save areas using the information contained in them to resume each procedure.

The formats of the stack frame save area and previous save area are shown in the CYBIL System Interface manual. For further information on the stack frame save area and previous save area, refer to volume II of the virtual state hardware reference manual.

#READ_REGISTER Function

The #READ_REGISTER function performs actions equivalent to the copy from state register (CPYSX) hardware instruction. It allows a program to read the contents of a process or processor register.

The format of the #READ_REGISTER function call is:

#READ_REGISTER(register_id)

register_id

An integer expression from 0 through 255 that identifies the number of the register to be read. Register numbers are given in the executive state hardware reference manual.

An integer value is returned.

The #WRITE_REGISTER procedure described in chapter 7 allows a program to change the contents of a process or processor register.

For further information on process and processor registers, and the CPYSX instruction, refer to volume II of the virtual state hardware reference manual.

#RING Function

The #RING function accepts a pointer and returns the integer value of the ring number contained in the pointer.

The format of the #RING function call is:

#RING(pointer)

pointer

Name of a direct pointer expression.

#SEGMENT Function

The #SEGMENT function accepts a pointer and returns the integer value of the segment number contained in the pointer.

The format of the #SEGMENT function call is:

#SEGMENT(pointer)

pointer

Name of a direct pointer expression.

User-Defined Functions

Function Declaration

You define your own functions with function declarations.

The format used for specifying a function is:

FUNCTION *{[attributes]}* **name** *{(formal_parameters)}* **: result_type;**†
  {declaration_list}
  statement_list
FUNCEND *{name}* ;

attributes

One or more of the following attributes. If more than one are specified, they are separated by commas.

XREF

The function has been compiled in a different module. In this case, the function declaration can contain the name and formal parameters, but no declaration list or statement list. In the other module, the function must have been declared with the XDCL attribute and an identical parameter list. If omitted, the function must be defined within the module where it is referenced.

XDCL

The function can be referenced from outside of the module in which it is located. This attribute can be included only in a function declared at the outermost level of a module; it cannot be contained in a program, procedure, or another function. Other modules that reference this function must contain the same function declaration with the XREF attribute specified.

#GATE††

Scope attribute that allows the function to be accessed by a procedure at a higher ring level. If #GATE is specified, the XDCL attribute is required also.

If no attributes are specified, the function is assumed to be in the same module in which it is called.

† Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a function declaration. If included in a CYBIL program run on NOS/VE, this parameter is ignored.

†† This attribute is not supported on variations of CYBIL available on other operating systems.

name
Name of the function. The function name is optional following FUNCEND.

formal_parameters
One or more parameters in the form:

VAR name {*,name*}... **: type**
{*,name* {*,name*}... *: type*}...

and/or:

name {*,name*}... **: type**
{*,name* {*,name*}... *: type*}...

The first form is called a reference parameter; the second form is called a value parameter. There is essentially no difference between them in the context of a function. However, procedures (and programs) do treat them differently. Both kinds of parameters can appear in the formal parameter list; if so, they are separated by semicolons (for example, I:INTEGER; VAR A:CHAR). Reference and value parameters are discussed in more detail later in this chapter under Parameter List.

result_type
The type of the result to be returned. It can be any fixed scalar, floating-point, pointer, or cell type.

declaration_list
Zero or more declarations.

statement_list
One or more statements.

In an assignment statement within a function, the lefthand side of the statement (the variable to receive the value) cannot be:

- A nonlocal variable.
- A formal parameter of the function.
- The object of a pointer variable.

User-defined functions cannot contain:

- Procedure call statements that call user-defined procedures or NOS/VE procedures.
- Parameters of type pointer to procedure.
- ALLOCATE, FREE, PUSH, or NEXT statements that have parameters that are not local variables.

Parameter List

A parameter list is an optional list of variable declarations that appears in the first statement of the function declaration. In the function declaration format shown earlier, they are shown as "formal_parameters". Declarations for formal parameters must appear in that first statement; they cannot appear in the declaration list in the body of the function.

A parameter list allows you to pass values from the calling program to the function. When a call is made to a function, parameters called actual parameters are included with the function name. The values of those actual parameters replace the formal parameters in the parameter list. Wherever the formal parameters exist in the statements within the function, the values of the corresponding actual parameters are substituted. For every formal parameter in a function declaration, there must be a corresponding actual parameter in the function call.

There are two kinds of parameters: reference parameters and value parameters. A reference parameter has the form:

VAR name *{,name}...* **: type**
{,name {,name}... : type}...

A value parameter has the form:

name *{,name}...* **: type**
{,name {,name}... : type}...

Procedures make a distinction between the two types of parameters, but functions do not. (In a procedure, the value of a reference parameter can change during execution of the procedure; a value parameter cannot change.) In a function, neither reference parameters nor value parameters can change in value. A formal reference parameter can be any fixed or adaptable type. A formal value parameter can be any fixed or adaptable type, except a heap or an array or record that contains a heap.

Reference parameters and value parameters can be specified in many combinations. When both kinds of parameters appear together, they must be separated by semicolons. Parameters of the same type can also be separated by semicolons instead of commas, but in this case, VAR must appear with each reference parameter. All of the following parameters lists are valid.

- `VAR i, j : integer; a, b : char;`
- `VAR i : integer; VAR j : integer; a : char; b : char;`
- `a : char; VAR i, j : integer; b : char;`
- `VAR i : integer, j : real; a : char, b : boolean;`

In each of the preceding examples, I and J are reference parameters; A and B are value parameters.

Referencing a Function

The call to the function is usually contained in an expression. The call consists of the function name (as given in the function declaration) and any parameters to be passed to the function in the following format:

name ({*actual_parameters*})

name

Name of the function.

actual_parameters

Zero or more expressions or variables to be substituted for formal parameters defined in the function declaration. If two or more are specified, they are separated by commas. They are substituted one-for-one based on their position within the list; that is, the first actual parameter replaces the first formal parameter, the second actual parameter replaces the second formal parameter, and so on. For every formal parameter in a function declaration, there must be a corresponding actual parameter in the function call.

If there were no formal parameters specified in the function declaration, there can be no actual parameters included in the function call. However, left and right parentheses are required to indicate the absence of parameters. In this case, the call is:

name( )

The function can be anywhere that a variable of the same type could be. The value returned by a function is the last value assigned to it. If control is returned to the calling point before an assignment is made, results are undefined.

The only types that can be returned as values of functions are the basic types: scalar, floating point, pointer, and cell.

Example:

The following function finds the smaller of two integer values represented by formal value parameters A and B. The smaller value is assigned to MIN, the name of the function, and that integer value is returned.

```
FUNCTION min (a, b : integer) : integer;
  IF a > b THEN
    min := b;
  ELSE
    min := a;
  IFEND;
FUNCEND min;
```

This function could be called using the following reference.

```
smaller := min(first,second);
```

The value of the variable FIRST is substituted for the formal parameter A; the value of SECOND is substituted for B. The value returned, the smaller value, replaces the entire function reference; the variable SMALLER is assigned the smaller value.

Procedures 7

This chapter descibes the procedures that are predefined in CYBIL and tells how you can define your own procedures.

Procedures 7

A procedure is one or more statements that perform a specific action and can be called by a single statement. A procedure allows you to associate a name with the statement list so that by specifying the name itself as if it were a statement, you cause the list to be executed. Declarations can be included and take effect when the procedure is called. A procedure call can optionally cause actual parameters included in the call to be substituted for the formal parameters in the procedure declaration before the procedure's statements are executed.

A procedure differs from a function in that:

- A procedure can, but does not always, return a value.
- The call to a procedure is the procedure's name itself; a function call by contrast must be part of an expression in a statement.
- There can be no value assigned to the procedure name as there is to a function name.

You can call procedures that are already defined in the language or you can define your own procedures. This chapter describes both.

Standard Procedures

The STRINGREP procedure described here is a standard CYBIL procedure. It can be used safely in variations of CYBIL available on other operating systems. The next section in this chapter, System-Dependent Procedures, describes procedures that may not always be available or that are unique to CYBIL on NOS/VE.

STRINGREP Procedure

The STRINGREP procedure converts one or more elements to a string of characters, then returns that string and the length of the string.

The format of the STRINGREP procedure call is:

STRINGREP(string_name, length, element {,*element*}...)

string_name

Name of a string type variable. (It can be specified as a substring.) The result is returned here. It will contain the character representations of the named element(s).

length

Name of an integer variable. Its value will be the length in characters of the resulting string variable, string_name. It will be less than or equal to the declared length of the string variable.

element

Name of the element to be converted. The element can be a scalar, floating-point, pointer, or string type. Formats for specifying particular types and rules for conversion of those types are discussed in more detail later in this chapter.

The named elements are converted to strings of characters. Those strings are then concatenated and returned left-justified in the named string variable. The length of the string variable is also returned. If the result of concatenating the string representations is longer than the length of the string variable, the result is truncated on the right; the length that will be returned is the length of the string variable.

Each individual element is converted and placed in a temporary field before concatenation with other elements. The length of the temporary field can be specified as part of the element parameter that is described in the following sections. Generally, numeric values are written right-justified in the temporary field with spaces added on the left to fill the field, if necessary. String or character values are written left-justified in the temporary field with spaces added on the right to fill the field, if necessary. For both numeric and alphabetic values, the field is filled with asterisk characters if it is too short to hold the resulting value. The value of the field length, when specified, must be greater than or equal to zero; otherwise, an error occurs.

The following paragraphs describe how the STRINGREP procedure converts specific types and how they appear in the temporary fields.

Integer Element

The format for specifying an integer element is:

expression { : *length* } { : *#(radix)* }

expression

An integer expression to be converted.

length

A positive integer expression specifying the length of the temporary field. The length must be greater than or equal to 2. If omitted, the temporary field is the minimum size required to hold the integer value and the leading sign character.

radix

Radix of expression. Possible values are 2, 8, 10, and 16. If omitted, 10 (decimal) is assumed.

The value of the integer expression is converted into a string representation in the desired radix. The resulting string representation is right-justified in the temporary field. If the expression is positive, a space precedes the leftmost significant digit. If the integer expression is negative, a minus sign precedes the leftmost significant digit. The leading space or hyphen must be considered a part of the length. (Thus, the length must be greater than or equal to 2 in order to hold the sign character and at least one digit.)

If a field length larger than necessary is specified, spaces are added on the left to fill the field. If the field length is not long enough to contain all digits and the sign character, the field is filled with a string of asterisk characters. If the field length is less than or equal to zero, an error occurs.

Character Element

The format for specifying a character element is:

expression { : *length* }

expression

A character expression to be converted.

length

A positive integer expression specifying the length of the temporary field. If omitted, a length of 1 is assumed.

A single character is left-justified in the temporary field. If a field length larger than necessary is specified, spaces are appended to the right to fill the field. Including a radix for a character element causes a compilation error.

Boolean Element

The format for specifying a boolean element is:

expression { : *length* }

expression

A boolean expression to be converted.

length

A positive integer expression specifying the length of the temporary field. If omitted, a length of 5 is assumed.

Either of the five-character strings '   TRUE' or 'FALSE' is left-justified in the temporary field. If a field length larger than necessary is specified, spaces are appended on the right to fill the field. If the field length is not long enough to contain all five characters, the temporary field is filled with asterisk characters. Including a radix for a boolean element causes a compilation error to occur.

Ordinal Element

The integer value of an ordinal expression is handled the same way as an integer element. Refer to the discussion under Integer Element earlier in this chapter.

Subrange Element

A subrange element is handled the same way as the element of which it is a subrange.

Floating-Point Element

The format for specifying a floating-point element is:

expression { : *length* { : *fraction* } }

expression

A real expression to be converted. If the value is INFINITE or INDEFINITE, an error occurs.

length

A positive integer expression specifying the length of the temporary field. If omitted, the temporary field is the minimum size required to hold the integer value and the necessary leading character.

fraction

Positive integer expression specifying the number of fractional digits to be included in a fixed-point format. Its value must be less than or equal to "length - 2". If omitted, conversion to floating-point format is assumed.

A floating-point expression can be converted into either a fixed-point format or a floating-point format depending on the fraction parameter. If it is included, the expression is converted to fixed-point format; if omitted, the expression is converted to floating-point format.

Fixed-Point Format

The form

expression {: *length*{: *fraction*}}

causes the specified expression to be converted to a string in fixed-point format. The string will have the specified length with the specified number of fractional digits to the right of the decimal place. The expression is rounded off so that the specified number of fractional digits are present. If no positive digit appears to the left of the decimal point, a 0 (zero) is inserted. When figuring the length required to hold the expression, the compiler counts all digits to the left of the decimal point (including 0 if it appears alone), the decimal point, and the specified number of fractional digits appearing to the right of the decimal point. If the expression is negative, an extra space is required for the minus sign. If a field length larger than necessary is specified, spaces are added on the left to fill the field. If the field length specified is not long enough to contain all digits, the sign character, and the decimal point, the field is filled with a string of asterisk characters.

Examples:

Value of Expression E	Format of Element	Resulting String
1.23456	E:6:2	`'  1.23'`
-1.23456	E:6:3	`'-1.235'`
0	E:5:2	`' 0.00'`

Floating-Point Format

The form

expression {: *length*}

causes the specified expression to be converted to a string in floating-point format.

The length of the temporary field is determined somewhat differently from the other elements. The system defines a maximum number of digits that can be contained in the mantissa of a real number and the number of digits that can be in the exponent. When the compiler figures the number of digits that will be in the mantissa, it first determines the number of spaces that must be present in the string. The number of digits in the exponent is required as are four additional spaces: one for the sign of the expression (a space if positive, - if negative), one for the decimal point in the mantissa, one for the exponent character (E), and one for the sign of the exponent (+ or -). The total number of required spaces is subtracted from the specified field length. The compiler then compares the result (field length minus required spaces) and the maximum number of digits allowed in the mantissa, and takes the smaller of the two. That number is used for the number of digits to be in the mantissa when the compiler rounds the floating-point expression.

If a field length larger than necessary is specified, spaces are added on the left to fill the field. If the fixed size of the exponent is larger than necessary, zeroes are filled in on the left. If the number that results from the subtraction of required spaces from the field length is less than 1, the field is filled with a string of asterisk characters.

Examples:

Value of Expression E	Format of Element	Resulting String
123.456	E:10	`' 1.23E+002'`
-123.456	E:11	`'-1.235E+002'`

Pointer Element

The format for specifying a pointer element is:

pointer { : *length* } { : *#(radix)* }

pointer

A pointer reference to be converted.

length

A positive integer expression specifying the length of the temporary field. If the field length is omitted, the temporary field is the minimum size required to contain the pointer value.

radix

Radix of the pointer value. Possible values are 2, 8, 10, and 16. For NOS/VE, the default radix is 16.

The value of the pointer expression is converted into a string representation in the specified radix. It is right-justified in the temporary field. If a field length larger than necessary is specified, spaces are added on the left to fill the field. If the field length is not long enough to contain all the digits, the field is filled with a string of asterisk characters.

String Element

The format for specifying a string element is:

expression { : *length* }

expression

A string variable, string constant, or substring to be converted.

length

A positive integer expression specifying the length of the temporary field. If omitted, the field is the minimum size required to contain the string expression.

A string expression is left-justified in the temporary field. If a field length larger than necessary is specified, spaces are appended on the right to fill the field. If the field length is shorter than the length of the string, the temporary field is filled with a string of asterisk characters.

System-Dependent Procedures

Of the procedures described here, some can be used only with NOS/VE; others may be available in variations of CYBIL on other operating systems, but they are not guaranteed to be. Keep in mind that programs using these procedures may not be transportable to other systems.

To use these procedures properly and efficiently, you should be familiar with basic hardware concepts of your computer system. This information can be found in volume II of the virtual state hardware reference manual.

The functions are described in alphabetic order according to the first alphabetic character.

#CALLER_ID Procedure

The #CALLER_ID procedure returns the identification (caller id) of the caller of a function or procedure. This procedure can be used only with NOS/VE.

The format of the #CALLER_ID procedure call is:

#CALLER_ID(id_record)

id_record

Name of the record that will contain the caller id information. It must be four bytes long.

The caller id is a record that contains the global/local key, ring number, and segment number of the caller. When a function or procedure is called, the caller id is placed in the leftmost 32 bits of the X0 register as a result of a call relative (CALLREL) or call indirect (CALLSEG) hardware instruction. The #CALLER_ID procedure accesses X0 while this information is there.

No special scope attributes (XDCL or XREF) are required in the calling function or procedure to use this procedure.

For further information on the caller id record and the CALLREL and CALLSEG instructions, refer to volume II of the virtual state hardware reference manual.

#COMPARE_SWAP Procedure

The #COMPARE_SWAP procedure performs actions equivalent to the compare swap (CMPXA) hardware instruction. It compares the contents of a variable with an expression. If the variable is unlocked and equal to the expression, the variable is swapped with a new expression. This procedure can be used only with NOS/VE.

The format of the #COMPARE_SWAP procedure call is:

#COMPARE_SWAP(lock_variable, initial_expression, new_expression, actual_variable, result_variable)

lock_variable

Name of the variable on which the compare swap operation is to be performed.

initial_expression

Expression that is compared to the lock variable. They must be equal for the swap operation to occur.

new_expression

Expression that specifies the value to be stored in the lock variable if the swap is successful (that is, the contents of lock_variable equals initial_expression).

actual_variable

Name of the variable into which the initial contents of the lock variable is returned. If the lock variable is locked, this field is not changed.

result_variable

Name of the variable into which the result of the compare swap instruction is returned. It must be a subrange from 0 through 2 where each value has the following significance.

Value	Significance
0	Swap operation was successful.
1	Swap operation failed because the initial expression was not equal to the contents of the lock variable.
2	Swap operation failed because the lock variable was locked.

The types of the lock variable, initial expression, new expression, and actual variable must be equivalent and have a size of eight bytes.

The lock variable is said to be locked if the leftmost 32 bits are ones. If it is locked, no action occurs. If it is unlocked, the contents of the lock variable is assigned to the actual variable. Then the lock variable is compared to an initial expression. If they are equal, a new expression is assigned to the lock variable. Otherwise, no swap occurs.

This procedure essentially performs the following statements.

```
IF (left half of lock_variable) = 0FFFFFFFF(16) THEN
   result_variable := 2;
ELSE
   actual_variable := lock_variable:
   IF lock_variable = initial_expression THEN
      lock_variable := new_expression;
      result_variable := 0;
   ELSE
      result_variable := 1;
   IFEND
IFEND
```

These statements are executed by the hardware as a noninterruptable sequence. Access to the lock_variable from other sources, such as another processor or peripheral processor (PP), is prevented while these statements are being executed.

For further information on the CMPXA instruction, refer to volume II of the virtual state hardware reference manual.

#CONVERT_POINTER_TO_PROCEDURE Procedure

The #CONVERT_POINTER_TO_PROCEDURE procedure converts a variable of the type pointer to procedure that has no parameters to a variable of the type pointer to procedure that can have parameters. This procedure may not be available on variations of CYBIL that execute on other operating systems.

The format of the #CONVERT_POINTER_TO_PROCEDURE procedure call is:

#CONVERT_POINTER_TO_PROCEDURE(pointer_1, pointer_2)

pointer_1

Name of a pointer to procedure variable with no parameters.

pointer_2

Name of a pointer to procedure variable with an arbitrary parameter list.

#HASH_SVA Procedure

The #HASH_SVA procedure performs actions equivalent to the load page table index (LPAGE) hardware instruction. This instruction searches the system page table (SPT) for a given system virtual address (SVA). This procedure can be used only with NOS/VE.

The format of the #HASH_SVA procedure call is:

#HASH_SVA(sva_variable, index, count, result_variable)

sva_variable

Name of the variable that contains the SVA for which the instruction will search.

index

Name of an integer variable that will contain a word index into the SPT. If the SVA is found, this index points to the SPT entry for the SVA. If the SVA is not found, it points to the last entry searched.

count

Name of an integer variable that will contain the number of SPT entries searched.

result_variable

Name of a boolean variable that is set to TRUE if the SVA is found.

The procedure returns either an index within the table if the SVA is found, or an index of the last entry searched if the SVA is not found. It also returns the number of entries searched and a boolean value indicating whether the entry was found.

For further information on the SVA, addressing in general, and the LPAGE instruction, refer to volume II of the virtual state hardware reference manual.

#KEYPOINT Procedure

The #KEYPOINT procedure generates an inline keypoint hardware instruction based on parameters supplied in the call. It allows performance monitoring of programs using keypoint instructions as trap interrupts. This procedure can be used only with NOS/VE.

The format of the #KEYPOINT procedure call is:

#KEYPOINT(class, data, identifier)

class

A constant integer expession from 0 through 15 that specifies the keypoint class. This value is placed in the j field of the hardware instruction.

data

A constant or variable expression from 0 through 0FFFFFFFF hexadecimal that specifies optional data to be collected with the keypoint. If it is the constant 0, a 0 is placed in the k field of the hardware instruction. If it is not 0, the value is placed in an X register and that register is placed in the k field of the hardware instruction.

identifier

A constant expression from 0 through 0FFFF hexadecimal that specifies a keypoint identifier. It is placed in the Q field of the hardware instruction.

For further information on the KEYPOINT instruction, refer to volume II of the virtual state hardware reference manual.

#PURGE_BUFFER Procedure

The #PURGE_BUFFER procedure performs actions equivalent to the purge hardware instruction. It purges the contents of cache or the map buffer. This procedure can be used only with NOS/VE. However, not all computer systems that support NOS/VE have cache and map buffers. If executed on a model without cache or map buffers, no action occurs.

The format of the #PURGE_BUFFER procedure call is:

#PURGE_BUFFER(option_value, address)

option_value
A constant integer expression from 0 through 15 that specifies one of the following purge options.

Value	Purge Option
0	Purge all entries in cache that are included in the 512-byte block defined by the system virtual address (SVA) in Xj.
1	Purge all entries in cache that are included in the active segment identifier (ASID) defined by the SVA in Xj.
2	Purge all entries in cache.
3	Purge all entries in cache that are included in the 512-byte block defined by the process virtual address (PVA) in Xj.
4–7	Purge all entries in cache that are included in the segment number defined by the PVA in Xj.
8	Purge all entries in the map (page table map if entries are kept in separate maps) relating to the page table entry defined by the SVA in Xj.

(Continued)

(Continued)

Value	Purge Option
9	Purge all entries in the map (page table map if entries are kept in separate maps) relating to the page table entries that are included in the segment defined by the SVA in Xj.
10 or A(16)	Purge all entries in the map (page table map if entries are kept in separate maps) relating to the page table entry defined by the PVA in Xj.
11 or B(16)	Purge all entries in the map (both the page table and segment map) relating to the segment table entry defined by the PVA in Xj, and to all page table entries included within that segment.
12–15 or C(16)-F(16)	Purge all entries in the map.

address

Name of a six-byte variable that specifies the PVA or SVA of the data to be purged.

For further information on addressing, cache and map buffers, and the purge instruction, refer to volume II of the virtual state hardware reference manual.

#SCAN Procedure

The #SCAN procedure scans a string from left to right until one of a specified set of characters is found or the entire string has been searched. This procedure may not be available on variations of CYBIL that execute on other operating systems.

The format of the #SCAN procedure call is:

#SCAN(scan_variable, string, index, result_variable)

scan_variable

Name of the variable that indicates the character values for which the string is scanned. The variable must be 256 bits long. Each bit of the variable represents the character in the corresponding position of the ASCII character set. If a bit is set, the corresponding character is one for which the procedure scans.

string

String or substring to be scanned.

index

Name of an integer variable. If a character is found during scanning, the index of that character is returned in this variable. The index of a character is that character's position in the string; for example, the index value of the first character is 1. If no matching values are found, the variable contains the string length plus 1.

result_variable

Name of a boolean variable, which is set to TRUE if the scan finds one of the selected characters.

The procedure looks for any one character from a set of characters specified in a 256-bit variable. Bits are set in the variable to correspond to the characters in the same positions in the ASCII character set collating sequence. A set bit indicates that the procedure scans the string for the corresponding character. The procedure stops if it finds one of the characters specified. It returns the position of the character that caused termination and the boolean variable that indicates whether a character was found.

#TRANSLATE Procedure

The #TRANSLATE procedure translates each character in a source field according to a translation table, and transfers the result to a destination field. This procedure may not be available on variations of CYBIL that execute on other operating systems.

The format of the #TRANSLATE procedure call is:

#TRANSLATE(table, source, destination)

table

Name of a string variable whose length is 256 characters. This variable defines the translation table.

source

String to be translated.

destination

Name of a string variable into which the translated string is transferred.

Translation of the string occurs from left to right with each source byte used as an index into the translation table. Translated bytes from the table are stored in the destination field.

If the length of the source field is less than the length of the destination field, translated spaces fill the destination field. If the source field is larger than the destination field, the rightmost characters of the source field are truncated.

#UNCHECKED_CONVERSION Procedure

The #UNCHECKED_CONVERSION procedure copies directly from a source field to a destination field. This procedure may not be available on variations of CYBIL that execute on other operating systems.

The format of the #UNCHECKED_CONVERSION procedure call is:

#UNCHECKED_CONVERSION(source, destination)

source

Name of a variable from which the copy is made.

destination

Name of a variable to which the copy is made.

The source and destination fields must have the same length in bits. Neither the source nor the destination field can be a pointer or contain a pointer. If either the source or destination field is the object of a pointer reference (pointer^), the pointer cannot be a pointer to a procedure.

The destination field must satisfy the same restrictions as the target of an assignment statement. This means that the destination field cannot be:

- A read-only variable.
- A formal value parameter of the procedure that calls the #UNCHECKED_CONVERSION procedure.
- A bound variant record.
- The tag field name of a bound variant record.
- A heap.
- An array or record that contains a heap.

#WRITE_REGISTER Procedure

The #WRITE_REGISTER procedure performs actions equivalent to the copy to state register (CPYXS) hardware instruction. It allows a program to change the contents of a process or processor register. This procedure can be used only with NOS/VE.

The format of the #WRITE_REGISTER procedure call is:

#WRITE_REGISTER(register_id, data)

register_id

An integer expression from 0 through 255 that identifies the number of the register to be written. Register numbers are given in the executive state hardware reference manual.

data

Integer expression that contains the data to be written to the register.

The #READ_REGISTER function described in chapter 6 allows a program to read the contents of a process or processor register.

Writing to certain registers requires special privileges. For further information on process and processor registers, and the CPYXS instruction, refer to volume II of the virtual state hardware reference manual.

User-Defined Procedures

Procedure Declaration

You define your own procedures with procedure declarations.

The format used for specifying a procedure is:

PROCEDURE {*[attributes]*} **name** {*(formal_parameters)*};†
  {*declaration_list*}
  {*statement_list*}
PROCEND {*name*};

attributes

One or more of the following attributes. If more than one are specified, they are separated by commas.

XREF

The procedure has been compiled in a different module. In this case, the procedure declaration can contain the name and formal parameters, but no declaration list or statement list. In the other module, the procedure must have been declared with the XDCL attribute and an identical parameter list. If omitted, the procedure must be defined within the module where it is called.

XDCL

The procedure can be called from outside of the module in which it is located. This attribute can be included only in a procedure declared at the outermost level of a module; it cannot be contained in a program, function, or another procedure. Other modules that call this procedure must contain the same procedure declaration with the XREF attribute specified.

† Some variations of CYBIL available on other operating systems allow an additional option, the alias name, in a procedure declaration. If included in a CYBIL program run on NOS/VE, this parameter is ignored.

INLINE

Instead of calling the procedure, the compiler inserts the actual procedure statements at the point in the code where the procedure call is made.

#GATE†

Scope attribute that allows the procedure to be accessed by a procedure at a higher ring level. If #GATE is specified, the XDCL attribute is required also.

If no attributes are specified, the procedure is assumed to be in the same module in which it is called.

name

Name of the procedure. The procedure name is optional following PROCEND.

formal_parameters

One or more parameters in the form:

VAR name {,*name*}...**: type**
{,*name* {,*name*}... : *type*}...

and/or:

name {,*name*}... **: type**
{,*name* {,*name*}... : *type*}...

The first form is called a reference parameter; its value can be changed during execution of the procedure. The second form is called a value parameter; its value cannot be changed by the procedure. Both kinds of parameters can appear in the formal parameter list; if so, they are separated by semicolons (for example, I:INTEGER; VAR A:CHAR). Reference and value parameters are discussed in more detail later in this chapter under Parameter List.

declaration_list

Zero or more declarations.

statement_list

Zero or more statements.

† This attribute is not supported on variations of CYBIL available on other operating systems.

Parameter List

A parameter list is an optional list of variable declarations that appears in the first statement of the procedure declaration. In the procedure declaration format shown earlier, they are shown as "formal_parameters". Declarations for formal parameters must appear in that first statement; they cannot appear in the declaration list in the body of the procedure.

A parameter list allows you to pass values from the calling program to the procedure. When a call is made to a procedure, parameters called actual parameters are included with the procedure name. The values of those actual parameters replace the formal parameters in the parameter list. Wherever the formal parameters exist in the statements within the procedure, the values of the corresponding actual parameters are substituted. For every formal parameter in a procedure declaration, there must be a corresponding actual parameter in the procedure call.

There are two kinds of parameters: reference parameters and value parameters. A reference parameter has the form:

VAR name {,*name*}... **: type**
 {,*name* {,*name*}... : *type*}...

When a reference parameter is used, the formal parameter represents the corresponding actual parameter throughout execution of the procedure. Thus, an assignment to a formal parameter changes the variable that was passed as the corresponding actual parameter. An actual parameter corresponding to a formal reference parameter must be addressable. A formal reference parameter can be any fixed or adaptable type. If the formal parameter is a fixed type, the actual parameter must be a variable or substring of an equivalent type. If the formal parameter is an adaptable type, the actual parameter must be a variable or substring whose type is potentially equivalent. (For further information on potentially equivalent types, refer to Equivalent Types in chapter 4.)

A value parameter has the form:

name {,*name*}... **: type**
{,*name* {,*name*}... : *type*}...

When a value parameter is used, the formal parameter takes on the value of the corresponding actual parameter. However, the procedure cannot change a value parameter by assigning a value to it or using it as an actual reference parameter to another procedure or function. A formal value parameter can be any fixed or adaptable type except a type that cannot have a value assigned, that is, a heap, or an array or record that contains a heap. If the formal parameter is a fixed type, the actual parameter can be any expression that could be assigned to a variable of that type. Strings must be of equal length. If the formal parameter is an adaptable type, the current type of the actual parameter must be one to which the formal parameter can adapt. If the formal parameter is an adaptable pointer, the actual parameter can be any pointer expression that could be assigned to the formal parameter. Both the value and the current type of the actual parameter are assigned to the formal parameter.

Reference parameters and value parameters can be specified in many combinations. When both kinds of parameters appear together, they must be separated by semicolons. Parameters of the same type can also be separated by semicolons instead of commas, but in this case, VAR must appear with each reference parameter. All of the following parameter lists are valid.

- `VAR i, j : integer; a, b : char;`
- `VAR i : integer; VAR j : integer; a : char; b : char;`
- `a : char; VAR i, j : integer; b : char;`
- `VAR i : integer, j : real; a : char, b : boolean;`

In each of the preceding examples, I and J are reference parameters; A and B are value parameters.

Calling a Procedure

A call to a procedure consists of the procedure name (as given in the procedure declaration) and any parameters to be passed to the procedure in the following format:

name {*(actual_parameters)*};

name

Name of the procedure or a pointer to a procedure.

actual_parameters

One or more expressions or variables to be substituted for formal parameters defined in the procedure declaration. If two or more are specified, they are separated by commas. They are substituted one-for-one based on their position within the list; that is, the first actual parameter replaces the first formal parameter, the second actual parameter replaces the second formal parameter, and so on. For every formal parameter in a procedure declaration, there must be a corresponding actual parameter in the procedure call.

A procedure is a type, like the types described in chapter 3. Procedure types are used for declaration of pointers to procedures; there are no procedure variables.

The lifetime of a formal parameter is the lifetime of the procedure in which it is a part. Storage space for the parameter is allocated when the procedure is entered and released when the procedure is left.

The lifetime of a variable that is allocated using the storage management statements (described in chapter 5) is the time between the allocation of storage (with the ALLOCATE statement) and the release of storage (with the FREE statement).

Two procedure types are equivalent if corresponding parameter segments have the same number of formal parameters, the same methods of passing parameters (reference or value), and equivalent types.

Example:

This example calculates the greatest common divisor X of M and N. M and N are passed as value parameters; that is, their values are used but M and N themselves are not changed. X, Y, and Z are reference parameters (preceded by the VAR keyword). Their original values have no meaning in the procedure; they are assigned new values in the procedure that destroy their previous values.

```
PROCEDURE gcd(m,n : integer; VAR x, y, z : integer);
  {Extended Euclid's Algorithm}
  VAR a1, a2, b1, b2, c, d, q, r : integer;

  a1 := 0;
  a2 := 1;
  b1 := 1;
  b2 := 0;
  c := m;
  d := n;

  WHILE d < > 0 DO
    {a1 * m + b1 * n = d, a2 * m + b2 * n = c}
    {gcd (c,d) = gcd (m,n)}
    q := c DIV d;
    r := c MOD d;
    a2 := a2 - q * a1;
    b2 := b2 - q * b1;
    c := d;
    d := r;
    r := a1;
    a1 := a2;
    a2 := r;
    r := b1;
    b1 := b2;
    b2 := r;
  WHILEND;

  x := c;
  y := a2;
  z := b2;
  {x = gcd (m,n), y * m + z * n = gcd (m,n,)}
PROCEND gcd;
```

The CYBIL Command and Other Compilation Facilities 8

This chapter describes the CYBIL command used to call the compiler, and the declarations, statements, and directives that can be used at compilation time.

The CYBIL Command and Other Compilation Facilities 8

This chapter describes the CYBIL command and the declarations, statements, and directives that can be used at compilation time. The CYBIL command is used to compile one or more CYBIL modules. The compilation statements and directives are used to construct the unit to be compiled and to control that process. If a compiler command and a directive specify conflicting options, the option encountered most recently is used.

For further information on program execution, refer to the SCL Object Code Management manual.

CYBIL Command

This command is a standard system command and uses the syntax and language elements for parameters described in the SCL Language Definition manual.

Purpose

The CYBIL command calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced.

Format

CYBIL
INPUT=file reference
LIST=file reference
BINARY_OBJECT=file reference
LIST_OPTIONS=list of keyword value
DEBUG=list of keyword value
ERROR_LEVEL=keyword value
OPTIMIZATION=keyword value
PAD=integer
RUNTIME_CHECKS=list of keyword value
STATUS=status variable

Parameters

INPUT (I)

Specifies the file that contains the source text to be read. Source input ends when an end-of-partition or an end-of-information is encountered on the source input file. If omitted, $INPUT is assumed.

LIST (L)

Specifies the file on which the compilation listing is to be written. If $NULL is specified, all compile-time output is discarded. If omitted, $LIST is assumed.

BINARY_OBJECT (B)

Specifies the file on which object code is to be written. If $NULL is specified, the compiler performs a syntactic and semantic scan of the program but does not generate object code. If omitted, $LOCAL.LGO is assumed.

LIST_OPTIONS (LO)

Specifies a combination of the following list options. If NONE is specified, no list options are selected. If omitted, option S (list the source input file) is assumed.

Option	Description
A	Produces an attribute list of source input block structure and relative stack. The attribute listing is produced following the source listing on the file specified by the LIST parameter or, if the LIST parameter is omitted, on file $LIST.
F	Produces a full listing. In effect, this option selects options A, S, and R.
O	Lists compiler-generated object code. When selected, this listing includes an assembly-like listing of the generated object code. This option has no effect if the BINARY_OBJECT parameter is set to $NULL.
R	Produces a symbolic cross-reference listing showing the location of a program entity definition and its use within a program.
RA	Produces a symbolic cross-reference listing of all program entities whether referenced or not.
S	Lists the source input file.
X	Used in conjunction with the compile-time directive LISTEXT so that listings can be externally controlled using the CYBIL command. The LISTEXT toggle must be ON. For further information, refer to Toggle Control under Compile-Time Directives later in this chapter.

DEBUG (D)

Specifies a combination of the following debug options. If omitted, symbol and line tables are generated.

Option	**Description**
DS	Compiles all debugging statements. A debugging statement is a statement in the source text that is ignored unless this option is specified. These statements are enclosed by the compile-time directives COMPILE and NOCOMPILE. (For further information, refer to Maintenance Control under Compile-Time Directives later in this chapter.) The symbol table and line table for interactive debugging are also generated.
NT	Symbol table and line table are not generated with the object code.

ERROR_LEVEL (EL)

Specifies one of the following error list options. If omitted, W (list warning and fatal diagnostics) is assumed.

Option	**Description**
F	Lists fatal diagnostics. If selected, only fatal diagnostics are listed.
W	Lists warning (informative) diagnostics as well as fatal diagnostics.

OPTIMIZATION (OPT)

Specifies one of the following optimization options. If omitted, DEBUG is assumed.

Option	Description
DEBUG	Object code is stylized to facilitate debugging. Stylized code contains a separate packet of instructions for each executable source statement; it carries no variable values across statement boundaries in registers, and it notifies Debug each time the beginning of a statement or procedure is reached.
LOW	Provides for keeping constant values in registers.
HIGH	Provides for keeping local variables in registers, passing parameters to local procedures in registers, and eliminating redundant memory references, common subexpressions, and jumps to jumps.

PAD

Generates the specified number of no-op (no operation) instructions between instructions that actually perform operations. If omitted, zero is assumed; no-op instructions are not generated.

RUNTIME_CHECKS (RC)

Specifies a combination of the following run-time checking options. If NONE is specified, no run-time checks are selected. If omitted, R (check ranges) and S (test array subscripts) are assumed.

Option	Description
N	Produces compiler-generated code that checks for a NIL value when a reference is made to the object of a pointer.
NONE	No run-time checks are produced.
R	Produces compiler-generated code to check ranges. Range checking code is generated for assignment to integer subranges, ordinal subranges, and character variables. All CASE statements are checked to ensure that the selection expression corresponds to one of the variant values specified if no ELSE clause is provided. All references to substrings are verified. If an offset (variable pointer) is specified on a RESET statement, it is checked to ensure that it is valid for the specified sequence.
S	Produces compiler-generated code to test the subscripting of arrays.

STATUS

Specifies that SCL will pass the compilation status to the specified variable. The compiler always returns a status variable indicating whether any fatal errors were found during the compilation that was just completed. You can test this status variable and take special action if fatal compilation errors occurred. If omitted and the status returned from the compiler is abnormal, SCL terminates the current command sequence.

Remarks If the compiler command specifies an option that differs from a directive, the latest occurrence of either the command or the directive takes precedence.

Example This command reads source code from a file named COMPILE, writes the compilation file on file LIST, and writes the object code on file BIN1. The listing includes source code, compiler-generated object code, and a symbolic cross-reference listing.

```
cybil i=compile l=list b=bin1 lo=(o,r)
```

Compilation Declarations and Statements

Many program elements defined in CYBIL have counterparts that can be used to control the compilation process. They include variable declarations, expressions, and the assignment and IF statements. The IF statement is used to specify certain areas of code to be compiled. The IF statement requires the use of expressions, which in turn require variables. Assignment statements are used to change the value of variables and, thus, expressions.

Compile-Time Variables

Only boolean type variables can be declared.

The format used to specify a boolean type compile-time variable is:

? VAR name {,*name*}... **: BOOLEAN := expression**
{, *name* {,*name*}... *: BOOLEAN := expression*}... **?;**

name

Name of the compile-time variable. This name must be unique among all other names in the program.

expression

A compile-time expression that specifies the initial value of the variable.

A compile-time declaration must appear before any compile-time variables are used. The scope of such a variable extends from the point at which it is declared to the end of the module. Compile-time variables can be used only in compile-time expressions and compile-time assignment statements.

Compile-Time Expressions

Compile-time expressions are composed of operands and operators like CYBIL-defined expressions. An operand can be:

- Either of the constants TRUE or FALSE.
- A compile-time variable.
- Another compile-time expression.

The operators are NOT, AND, OR, and XOR. Their order of evaluation from highest to lowest is:

- NOT
- AND
- OR and XOR

These operators have their usual meanings, as described under Operators in chapter 5.

Compile-Time Assignment Statement

A compile-time assignment statement assigns a value to a compile-time variable.

The format of the compile-time assignment statement is:

? name := expression ?;

name

Name of a compile-time variable.

expression

A compile-time expression.

Compile-Time IF Statement

The compile-time IF statement compiles or skips a certain area of code depending on whether a given expression is true or false.

The format of the compile-time IF statement is:

? IF expression THEN
  code
 { *? ELSE*
  code }
? IFEND

expression

A boolean compile-time expression.

code

An area of CYBIL code or text.

When the expression is evaluated as true, the code following the reserved word THEN is compiled. When compilation of that code is completed, compilation continues with the first statement following IFEND. When the expression is false, compilation continues following the ELSE phrase, if it is included, or following IFEND.

The ELSE clause is optional. If included, the ELSE clause designates an area of code that is compiled when the preceding expression is false.

Example:

The following example shows the declaration of a compile-time variable named SMALL_SIZE that is initialized to the value TRUE. A line of CYBIL code declaring an array named TABLE is compiled. Then a compile-time IF statement checks the value of SMALL_SIZE. If it is TRUE, the line of CYBIL code calling a procedure named BUBBLESORT is compiled in the program. If it is FALSE, the CYBIL line calling procedure QUICKSORT is inserted instead. Because SMALL_SIZE was initialized to TRUE, the call to BUBBLESORT is included in the compiled program.

```
? VAR small_size : boolean := TRUE ?;

VAR table : array [1..50] of integer;

? IF small_size = TRUE THEN

  bubblesort (table);

? ELSE

  quicksort (table);

? IFEND
```

Compile-Time Directives

Compile-time directives allow you to perform the following activities during compilation.

- Set toggles that turn on or off listing options such as source code listing and object code listing (SET, PUSH, POP, and RESET directives when they contain one or more of the listing options).
- Set toggles that turn on or off run-time options such as range checking and array subscript checking (SET, PUSH, POP, and RESET directives when they contain one or more of the run-time checking options).
- Specify the layout of the source text to be used (LEFT and RIGHT margin directives).
- Specify the layout of the resulting listing (EJECT, SPACING, SKIP, NEWTITLE, TITLE, and OLDTITLE directives).
- Specify what code to compile (COMPILE and NOCOMPILE directives).
- Include comments in the object module (COMMENT directive).

You can specify one or more directives with the format:

?? directive {,*directive*}... **??**

directive

One of the directives discussed in the remainder of this chapter. They can be broken down into four categories:

- Toggle control (SET, PUSH, POP, and RESET)
- Layout control (LEFT, RIGHT, EJECT, SPACING, SKIP, NEWTITLE, TITLE, and OLDTITLE)
- Maintenance control (COMPILE and NOCOMPILE)
- Object code comment control (COMMENT)

Directives must be bounded by a pair of consecutive question marks. These delimiters are not shown in the following formats for individual directives, but they are required around one or more directives.

If a directive differs from an option specified on a compiler command, the latest occurrence of either the directive or the command takes precedence.

Toggle Control

Toggle controls can set the values of individual toggles, save and restore preceding toggle values in a last in–first out manner, and reset all toggles to their initial values.

SET Directive

The SET directive specifies the setting of one or more toggles.

The format of the SET directive is:

SET (toggle_name := condition {,*toggle_name := condition*}...)

toggle_name

Name of the toggle being set. Listing toggles are described in table 8-1. Run-time checking toggles are described in table 8-2. The names of toggles can be used freely outside of directives.

condition

ON or OFF. If a toggle is ON, the activity associated with it is performed during compilation; if it is OFF, the activity is not performed.

All settings specified in the SET directive are performed together. If the directive list contains more than one setting for a single toggle, the rightmost setting in the list is used.

PUSH Directive

The PUSH directive specifies the setting of one or more toggles like the SET directive, but before the settings are put into effect, a record of the current state of all toggles is saved for later use.

The format of the PUSH directive is:

PUSH (toggle_name := condition {,*toggle_name* := *condition*}...)

toggle_name

Name of the toggle being set. Listing toggles are described in table 8-1. Run-time checking toggles are described in table 8-2. The names of toggles can be used freely outside of directives.

condition

ON or OFF. If a toggle is ON, the activity associated with it is performed during compilation; if it is OFF, the activity is not performed.

Settings in the PUSH list are performed in the same manner as a SET list. If the directive list contains more than one setting for a single toggle, the rightmost setting in the list is used.

The POP directive, described later in this chapter, restores the original toggle settings in a last in–first out manner (that is, the last record to be saved is the first to be restored).

Example:

This example turns off listing temporarily, that is, until the POP directive is encountered.

```
?? PUSH (LIST := OFF) ??
     .
     .
     .
?? POP ??
```

Table 8-1 describes the listing toggles and gives their initial settings.

Table 8-1. Listing Toggles

Toggle	Initial Value	Description
LIST	ON	Determines whether other listing toggles are read. When ON, a source listing is produced and the other listing toggles are used to control other aspects of listing. When OFF, no listing is produced; the other listing toggles are ignored.
LISTOBJ	OFF	Controls the listing of generated object code. When ON, object code is interspersed with source code following the corresponding source code line.
LISTCTS	OFF	Controls the listing of the listing toggle directives and layout directives.
LISTEXT	OFF	When ON, the listing of source statements is controlled by a list option on the CYBIL compiler command.
LISTALL	Not applicable	This option represents all of the listing toggles. When ON, all other listing toggles are ON; when OFF, all other listing toggles are OFF.

Table 8-2 describes the run-time checking toggles and gives their initial settings.

Table 8-2. Run-Time Checking Toggles

Toggle	Initial Value	Description
CHKRNG	ON	Controls the generation of object code that performs range checking of scalar subrange assignments and case variables.
CHKSUB	ON	Controls the generation of object code that checks array subscripts (indexes) and substring selections to verify that they are valid.
CHKNIL	OFF	Controls the generation of object code that checks for a NIL value when a reference is made to the object of a pointer.
CHKALL	Not applicable	This option represents all run-time checking toggles. When ON, all other run-time checking toggles are ON; when OFF, all other run-time checking toggles are OFF.

POP Directive

The POP directive restores the last toggle settings that were saved by the PUSH directive.

The format of the POP directive is:

POP

If no record was kept (such as when a SET directive is performed), the initial settings are restored.

Example:

This example shows a PUSH directive that temporarily turns off listing. The POP directive restores listing.

```
?? PUSH (LIST := OFF) ??
    .
    .
    .
?? POP ??
```

RESET Directive

The RESET directive restores the initial toggle settings.

The format of the RESET directive is:

RESET

When the RESET directive is performed, any record of previous settings is destroyed.

Layout Control

Layout controls are used to specify the margins of the source text and to control the layout of the listing.

LEFT and RIGHT Directives

The LEFT and RIGHT directives specify the column number of the left and right margins of the source text, respectively.

The formats of the LEFT and RIGHT directives are:

LEFT := integer

RIGHT := integer

integer

An integer value that represents the column number of the left and right margins, respectively.

The left margin must be greater than zero; that is,

left margin > 0

The right margin must be greater than or equal to the left margin plus 10, and less than or equal to 110; that is,

left margin + 10 <= right margin <= 110

All source text left of the left margin and right of the right margin is ignored.

If the margin directives are not used, the left margin is assumed to begin in column 1 with the right margin in column 79.

Example:

This example sets the left margin at column 1 and the right margin at column 110.

```
?? LEFT := 1, RIGHT := 110 ??
```

EJECT Directive

The EJECT directive causes the paper to be advanced to the top of the next page.

The format for specifying the EJECT directive is:

EJECT

SPACING Directive

The SPACING directive specifies the number of blank lines between individual lines of the listing.

The format of the SPACING directive is:

SPACING := spacing

spacing

One of the values 1, 2, or 3 specifying single, double, and triple spacing, respectively.

An undefined value has no effect on spacing, but an error message is issued.

If the SPACING directive is not used, single spacing (no intervening blank lines) is assumed.

SKIP Directive

The SKIP directive specifies that a given number of lines is to be skipped.

The format of the SKIP directive is:

SKIP := lines

lines

Integer value specifying the number of lines to skip. It must be greater than or equal to 1.

If the number of lines specified is larger than the number of lines on the page, or if it would cause the paper to skip past the bottom of the current page, the paper is advanced to the top of the next page.

NEWTITLE Directive

The NEWTITLE directive specifies a new, additional title to be used on a page while saving the current title.

The format of the NEWTITLE directive is:

NEWTITLE := 'character_string'

character_string

A character string specifying the title to be used. A single quote mark is indicated by two consecutive quote marks enclosed by quote marks (that is, '''').

The current title is saved and the given character string becomes the current title. A standard page header is always the first title printed on a page, followed by user-defined titles in the order in which they were saved. This means that titles are saved and restored in a last in–first out order, but they are printed in a first in–first out order. There is always a single empty line between the standard page header and any user-defined titles. There is always at least one empty line between the last title and the text.

The maximum number of titles that can be specified is 10. Any attempts to add more titles is ignored.

Titling does not take effect until the top of the next printed page.

TITLE Directive

The TITLE directive replaces the current user-defined title with the given character string.

The format of the TITLE directive is:

TITLE := 'character_string'

character_string

A character string specifying the title to be used. A single quote mark is indicated by two consecutive quote marks enclosed by quote marks (that is, '''').

If there is no user-defined title currently, the character string becomes the current title.

A standard page header is always the first title printed on a page. There is always a single empty line between the standard page header and any user-defined titles. There is always at least one empty line between the last title and the text.

Titling does not take effect until the top of the next printed page.

OLDTITLE Directive

The OLDTITLE directive restores the last user-defined title that was saved, making it the current title.

The format of the OLDTITLE directive is:

OLDTITLE

If there is no saved title, no action occurs.

Maintenance Control

COMPILE Directive

The COMPILE directive causes compilation to occur, or to resume after the occurrence of a NOCOMPILE directive.

The format of the COMPILE directive is:

COMPILE

If neither the COMPILE nor NOCOMPILE directive is used, the COMPILE directive is assumed; source code is compiled.

When the CYBIL command includes the Debug parameter with DS specified, debugging statements enclosed by the COMPILE and NOCOMPILE directives are compiled.

NOCOMPILE Directive

The NOCOMPILE directive causes compilation to stop until the occurrence of a COMPILE directive or the end of the module.

The format of the NOCOMPILE directive is:

NOCOMPILE

NOCOMPILE continues listing source code and text according to the listing toggles and layout directives, interpreting and obeying directives, but source code is not compiled until a COMPILE directive is encountered or a MODEND statement is encountered.

When the CYBIL command includes the Debug parameter with DS specified, debugging statements enclosed by the COMPILE and NOCOMPILE directives are compiled.

Comment Control

COMMENT Directive

The COMMENT directive causes the compiler to include the given character string in the commentary portion of the object module generated by the compilation process.

The format of the COMMENT directive is:

COMMENT := 'character_string'

character_string

A character string of up to 40 characters that specifies a compile-time comment.

This directive allows you to include comments in object modules so that the comments appear in the load maps. Any number of comments can be included, but only the last comment encountered appears.

Example:

```
?? COMMENT := 'Copyright 1983 by Control Data Corporation' ??
```

The Debug Utility 9

This chapter describes the Debug utility, which aids in debugging CYBIL programs.

The Debug Utility 9

Introduction

The Debug utility provides source code level symbolic debugging for programs written in COBOL, CYBIL, and FORTRAN, and machine code level debugging for object modules. Using Debug does not require source-level program modifications, a knowledge of assembly language, or the ability to interpret extensive memory dumps. Debugging can be done at the source language level.

Debug enables you to monitor and control the execution of programs in interactive and batch mode. Debug allows program conditions to be modified and tested during execution. With Debug, you can:

- Suspend program execution at specified locations, such as line 398 of module MAIN_PROGRAM.
- Suspend program execution when a selected event occurs, such as writing to a specified location.
- Display and change the values of program variables, memory locations, and registers while execution is suspended.
- Display the procedure calls that led to the current location (call traceback information).
- Display the environment that you are currently debugging under.
- Resume program execution at the location where execution was suspended or at another location.
- Step through a program by lines or procedures.

Because Debug is a command utility, SCL features are available while Debug is in control. With SCL, you can:

- Temporarily read commands from a file other than the Debug input file using the SCL command INCLUDE_FILE.
- Enter multiple commands, separated by semicolons, on one line.
- Continue a single command on one or more continuation lines.
- Evaluate and display SCL expressions using the SCL command DISPLAY_VALUE.
- Echo Debug commands to one or more files, and write Debug output to several files using the SCL command CREATE_FILE_CONNECTION.
- Include Debug commands in SCL procedures.
- Enter commands for processing by another active command processor, such as an editor to examine your source listing.

Accessing Debug

You can access Debug explicitly when executing your program. You can also access Debug when your program aborts unexpectedly.

Accessing Debug During Program Execution

Every program has various attributes that control its execution. Among these are the Debug attributes DEBUG_MODE, DEBUG_INPUT, and DEBUG_OUTPUT. These attributes are defined as follows:

DEBUG_MODE = ON or OFF

A keyword value that determines whether or not the program is to be executed under control of Debug.

DEBUG_INPUT = file

The file from which Debug initially reads commands when DEBUG_MODE=ON.

DEBUG_OUTPUT = file

The file to which Debug initially writes its output.

These attributes can be specified at two levels: program level and job level. Program level specifications apply to a specific program. Job level specifications apply to all programs that do not explicitly specify values at the program level.

Program level specifications are set as parameter values on the SCL command EXECUTE_TASK or on the SCL CREATE_OBJECT_LIBRARY utility's subcommand CREATE_PROGRAM_DESCRIPTION. Job level specifications are set as parameter values of the SCL command SET_PROGRAM_ATTRIBUTES. (Refer to the SCL Language Definition manual and the SCL Object Code Management manual for complete descriptions of these commands.)

For example, if you issue

```
set_program_attributes debug_mode=on
```

just after logging in, all program executions will be under control of Debug unless you specify DEBUG_MODE=OFF on the EXECUTE_TASK command or in a previously created program description. You can change job level attributes at any time by issuing another SET_PROGRAM_ATTRIBUTES command.

Initially, the values of the job level Debug attributes are DEBUG_MODE=OFF, DEBUG_INPUT=COMMAND, and DEBUG_OUTPUT=$OUTPUT. For interactive jobs, COMMAND and $OUTPUT are assigned to the terminal by default.

Individual sites and individual users at a site can change these initial defaults by including a SET_PROGRAM_ATTRIBUTES command in the system or user prologue file.

Accessing Debug When Program Failure Occurs

Once you have a working program, you generally want to access Debug only if the program unexpectedly fails. The program attributes that control Debug when a working program fails are ABORT_FILE and DEBUG_OUTPUT. These attributes are defined as follows:

ABORT_FILE = file

The file from which Debug initially reads commands if the program aborts when DEBUG_MODE=OFF.

DEBUG_OUTPUT = file

The file to which Debug initially writes its output.

These attributes can be specified at two levels: program level and job level. Program level specifications apply to a specific program. Job level specifications apply to all programs that do not explicitly specify values at the program level.

Program level specifications are set as parameter values on the SCL command EXECUTE_TASK or on the SCL CREATE_OBJECT_LIBRARY utility's subcommand CREATE_PROGRAM_DESCRIPTION. Job level specifications are set as parameter values of the SCL command SET_ PROGRAM_ATTRIBUTES. (Refer to the SCL Language Definition manual and the SCL Object Code Management manual for complete descriptions of these SCL commands.)

For example, if you issue

```
set_program_attributes debug_mode=off, abort_file=abortfile
```

just after logging in, Debug will not gain control unless the program fails. Programs will not execute under the control of Debug unless you specify DEBUG_MODE=ON on the EXECUTE_TASK command or in a previously created program description. You can change job level attributes at any time by issuing another SET_PROGRAM_ATTRIBUTES command.

The initial value of ABORT_FILE is $NULL, the special system file with no data in it. DEBUG_MODE must be off and ABORT_FILE must be a file other than $NULL.

Debug Concepts

This section contains miscellaneous information that applies to Debug usage. This information includes Debug input/output, status variable, breaks, and addressing.

Debug Input/Output

Although Debug input/output takes place automatically, you can, by manipulating the Debug input/output files, expand the capabilities of Debug.

Debug Input

Debug commands are initially read from the file specified by the DEBUG_ INPUT parameter or the ABORT_FILE parameter of the SCL commands EXECUTE_TASK, CREATE_PROGRAM_DESCRIPTION, or SET_ PROGRAM_ATTRIBUTES commands. For interactive jobs, the default input file COMMAND is the terminal. For batch jobs, COMMAND is the stream of lines being read. You cannot use COMMAND as the source of Debug input for a batch job because COMMAND is positioned at beginning-of-information, which is your login. Instead, you must copy the Debug input to another file and use that as the Debug input.

You can change the input file temporarily by entering an SCL INCLUDE_ FILE command. As soon as the command is entered, commands are read from the specified file until an end-of-partition, an end-of-information, or a RUN command is encountered. If an end-of-partition or an end-of-file is encountered, commands are again read from the file that contained the INCLUDE_FILE command. If a RUN command is encountered, program execution is resumed; any remaining commands in the file that was included are not processed. When Debug again gains control, commands are read from the current Debug input file.

The Debug command CHANGE_DEFAULT (described in detail later in this chapter) can also be used to change the Debug command source. The DEBUG_INPUT parameter of the CHANGE_DEFAULT command changes the command source so that Debug commands are read from the specified file when Debug gains control after program execution has been resumed. Unlike the INCLUDE_FILE command, the CHANGE_DEFAULT command has no effect on the current command source.

The default Debug input file is COMMAND. In interactive jobs, COMMAND is the terminal. In batch jobs, it is the normal command stream. If Debug is activated from within an SCL procedure, commands are read from COMMAND when Debug gains control, not from the procedure. To force Debug to read commands from the procedure, specify

```
debug_input = $command
```

in the program description or on the EXECUTE_TASK command.

Debug Output

Debug output (messages and information produced by Debug display commands) is initially written to the file specified by the DEBUG_OUTPUT parameter (default output file is $OUTPUT) of the SCL commands EXECUTE_TASK, CREATE_PROGRAM_DESCRIPTION, or SET_PROGRAM_ATTRIBUTES. The OUTPUT parameter of the Debug display commands can be used to divert display output to another file; the diversion applies only to the command that contains the OUTPUT parameter.

The Debug command CHANGE_DEFAULT (described in detail later in this chapter) can be used to change the current Debug output file. The DEBUG_OUTPUT parameter of the CHANGE_DEFAULT command causes Debug to write all output to the specified file; the change takes place as soon as the command is executed.

The default Debug output file is $OUTPUT. $OUTPUT is the terminal for interactive jobs and the listing file for batch jobs. Initially, $OUTPUT is connected to the actual file OUTPUT. You can connect $OUTPUT to other files by using the SCL command CREATE_FILE_CONNNECTION. If the standard files $ECHO, $RESPONSE, and $ERRORS are also connected to one of the actual output files, a complete record of a Debug session can be created.

Status Variable

All Debug commands have an optional parameter called STATUS. When you specify this parameter, a previously declared SCL variable of kind STATUS must be supplied as its value. (Refer to the SCL Language Definition manual for a discussion of SCL variables.) This variable contains the completion status of the command.

A status variable is a record that contains the following fields:

NORMAL

A boolean that has a value of FALSE if the command could not be processed correctly and a value of TRUE if the command was processed correctly.

IDENTIFIER

A string with a length of 2 that contains the product identifier of the processor in which the error was detected. The product identifier for Debug is DB. This field is undefined when the command is processed correctly.

CONDITION

An integer code that identifies the detected error. The two leftmost digits in a Debug condition code are 64. This field is undefined when the command is processed correctly.

TEXT

A string with a length of 256 that contains the error message text. This field is undefined when the command is processed correctly.

The presence of the STATUS parameter on a command causes the next command to be processed even if an error condition was encountered. After checking the contents of the status variable, you can use succeeding commands to alter the flow of control based upon the occurrence of error conditions.

Breaks

The primary mechanism for Debug to gain control from an executing program is the user-defined break. A user-defined break specifies one or more events and an address range such that when a specified event occurs within the address range, program execution is interrupted and Debug takes control. Many events can be specified, for example, when execution reaches a specific place, before a branch to a specific address range occurs, or before a write into memory. Address ranges also can be specified in many forms. You cannot set two breaks for the same event at the same address range or overlapping address ranges. Once set, a break stays set until it is explicitly deleted or implicitly deleted with the DELETE_BREAK ALL command. The SET_BREAK, DELETE_BREAK, and DISPLAY_BREAK commands are used to set, delete, and display break definitions. (These commands are described in detail later in this chapter.)

The maximum number of breaks that the Debug utility can handle is 64. Of these 64 breaks, 32 can be the type of break that is detected by Debug hardware (read, write, call, branch, execution, and read next instruction). Some breaks that you set cause Debug to set one or more internal breaks. Thus, the actual maximum number of breaks that are available to you is not a fixed number. A message is issued when another break cannot be set.

Addressing

Debug uses source level addresses when addresses are reported in Debug command output, such as when DISPLAY_CALL or DISPLAY_BREAK is executed and when Debug gains control. Debug also uses source level addresses when addresses are referenced in Debug commands, such as SET_BREAK and DISPLAY_MEMORY.

Reported Addresses

The level of reported addresses is determined by the information available. For CYBIL programs, the following are available by default:

- Module address tables indicating where modules are located.
- Line address tables indicating where code for each line is located.
- Symbol tables indicating where the value of each program name is located.

If you specify DEBUG=NT on the CYBIL command, however, line address and symbol table addresses are suppressed. In this case, only module and machine level addressing are possible.

Addresses in the message issued when Debug gains control (the break report message) are formatted as follows depending on the information available.

When line and module tables are available (symbolic addressing):

If the address corresponds to the beginning of a line, then

M=module_name L=line_number

otherwise, if the address is somewhere within the line, then

M=module_name L=line_number BO=byte_offset_from_
start_of_line

When only the module table is available (module addressing):

If the module is not bound (refer to the discussion of bound modules later in this chapter), then

M=source_module_name P=procedure_name BO=byte_
offset_from_start_of_procedure

otherwise, if the module is bound, then

M=source_module_name BO=byte_offset_from_
start_of_bound_module

When line and module tables are not available (machine addressing):

A=machine_address

Within the address formats:

- module_name and procedure_name correspond to the source program module and procedure names.
- line_number corresponds to a line number on the source listing.
- byte_offset is a decimal number corresponding to the number of bytes beyond the beginning of a line or a hexadecimal number corresponding to the number of bytes beyond the start of a procedure or bound module.
- machine_address is a set of three hexadecimal numbers representing the ring number, segment number, and segment offset of a machine address.

Addresses reported in command output also provide the highest address level possible, but they are not always formatted the same as in break report messages. Addresses shown in DISPLAY_BREAK output are very similar, but addresses shown in DISPLAY_CALL output contain both the procedure name and line number. Typical DISPLAY_CALL output might look like the following:

```
--  Traceback from procedure PROC2 module MOD2 at line 34
--  Called from procedure PROC1 module MOD2 at line 55 byte
    offset 4
--  Called from procedure BEGIN_PROCESS module MOD1 byte
    offset 1A3(16)
```

Addresses shown in DISPLAY_REGISTER output are formatted only as hexadecimal addresses in the form

r sss oooooooo

where r is the ring number, sss is the segment number, and oooooooo is the offset from the start of the segment. Pointer addresses displayed by DISPLAY_PROGRAM_VALUE are also formatted as hexadecimal machine addresses except for pointers to procedures; dereferenced pointers to procedures are displayed as the procedure name if possible.

Referenced Addresses

Several Debug commands reference program code and data addresses. For example, SET_BREAK designates an address or address range for break events, DISPLAY_MEMORY specifies the address of memory to be displayed, and DISPLAY_PROGRAM_VALUE names a program identifier whose value is to be displayed.

Just as for reporting addresses, the capabilities available when referencing program addresses depend on the information available:

- Symbolic addressing (source level addressing) is available if line and symbol tables exist (they exist unless line number and symbol table generation is specifically turned off at compile time).
- Module/procedure offset addressing is available if module tables exist (they always do for user programs).
- Machine-level addressing is always available.

Addresses can be referenced in many more forms than the form in which they are reported. For example, entry point names, section names, and program names can be referenced, but addresses are never reported in these terms. Machine level addresses can be referenced only as a single integer (a 12-digit hexadecimal value); they are reported, however, either as a 12-digit hexadecimal integer or as three separate integers corresponding to ring number, segment number, and byte offset from the start of the segment.

Not all address forms, however, are used by all commands. For example, the DISPLAY_PROGRAM_VALUE command allows a program name to be referenced by name, including all of the subscripting and qualification syntax. But, the DISPLAY_PROGRAM_VALUE command does not allow machine level addressing. The DISPLAY_MEMORY command, on the other hand, allows machine and module addressing but almost no symbolic level addressing. The SET_BREAK command allows all forms except names defined in a source program.

The different forms of addresses are specified by different parameters. LINE, MODULE, PROCEDURE, NAME, ENTRY_POINT, SECTION, and ADDRESS are typical address parameter names. Many of these address parameters can be used in combination to specify an address. For example, LINE and MODULE together specify a particular line of a particular module. NAME, MODULE, and PROCEDURE together specify a particular name of a particular procedure in a particular module. Similarly, SECTION can be used in conjunction with MODULE. ENTRY_POINT and ADDRESS, however, cannot be used in conjunction with MODULE or with each other because they specify addresses independent of any module. Debug issues an error message if an invalid combination of address parameters is used.

The BYTE_OFFSET parameter can be used to modify the address parameters. For example, the MODULE parameter without the BYTE_OFFSET parameter specifies the first byte of the module; the MODULE parameter modified with BYTE_OFFSET=4, on the other hand, specifies the fifth byte of the module.

Another parameter, BYTE_COUNT, can be used to establish the block size (address range) associated with a referenced address. The BYTE_COUNT parameter indicates how many memory bytes are to be included in the block. For example,

```
section=trap, byte_count=3
```

identifies a three-byte block that begins at section TRAP. BYTE_COUNT and BYTE_OFFSET can be used to modify any referenced address except a program name (NAME parameter).

Addressing Bound Modules

Individual modules can be bound (combined) to form a new load module that loads and executes faster than the original separate modules. (For further information, refer to the CREATE_OBJECT_LIBRARY command in the SCL Object Code Management manual.) Binding modules together has no effect on address reporting or address referencing at the symbolic level; you can debug bound modules in terms of their component module names, line numbers, and identifier names.

Binding does, however, have an effect on module/procedure and module/section offset addressing. After binding, original module and procedure names are not available when the tables that support symbolic addressing are not available; addresses are reported and must be referenced in terms of the new bound module name and byte offsets from the beginning of the module. Code from all original component modules is combined into one code section, static data from all original modules are combined into one static data memory section, and so forth, such that the original component portions of each section cannot be distinguished by Debug. You can deduce where each component portion is by inspecting the section map produced by the GENERATE_LIBRARY subcommand (described in the SCL Object Code Management manual).

Debugging Optimized Code

Most compilers can generate more than one level of object code. The OPTIMIZATION parameter on the compiler call controls the level of object code optimization. The default value is OPTIMIZATION=DEBUG, which generates the most debuggable object code possible. This level of object code contains a separate packet of machine instructions for each executable source statement, carries no altered variable values across statement boundaries in registers without also updating their values in memory, enables Debug to recognize that start of execution of each new line or procedure, and ensures that Debug can always find actual parameter lists.

If some higher level of optimization is selected, Debug can still function, but with restricted capabilities. For example, you cannot display program identifier values that are permanently allocated to machine registers. When values are temporarily carried in registers between statements, or when code for several source statements is mixed together, displayed values may not be the most recent values. Break report locations may not be as precise either.

Debugging With Condition Handlers

Condition handlers are special procedures whose purpose is to process conditions, or exceptions, when they arise. They are automatically activated by NOS/VE when the conditions for which they have been established occur. Condition handlers can be established for one or more classes of conditions. Refer to the CYBIL System Interface manual for a complete discussion of how to write condition handlers.

When executing with DEBUG_MODE=ON, Debug first gains control when any condition occurs, except job resource conditions, detected uncorrected error conditions, and block exit conditions. The condition handler of the program, if one exists, is not executed until a Debug RUN command is executed.

The condition handler of the program can be debugged using Debug, but the program will not execute until you have had a chance to respond to the condition. For conditions for which breaks can be set, a RUN command can be associated with the break so that the command is automatically executed when the break occurs. (Refer to the COMMAND parameter of the SET_ BREAK command later in this chapter.) This mechanism makes it possible to effectively circumvent the preemptive control of Debug. It appears as though Debug did not get control since the RUN command automatically executes the instant the condition arises.

Interrupt Processing While Debugging

Three external events can interrupt an executing user program or the Debug utility. These events are pause break, terminate break, and nearly exhausted resource. Table 9-1 shows the effects of these interrupts.

Table 9-1. Effects of Interrupts While Debugging

Interrupt	User Program Executing	Debug Executing
Pause Break	Debug gains control, reports where the program was executing when the interrupt occurred, and prompts for commands.	Default system action occurs. If you have established a handler for this condition, that handler gains control. Debug does not gain control unless the handler returns with normal status.
Terminate Break	Debug gains control, reports where the program was executing when the interrupt occurred, and prompts for commands.	If a Debug command is executing, that command is terminated and you are prompted for a new command. If Debug is already waiting for a command, the terminate break is ignored.
Nearly Exhausted Resource	Debug does not get control. The user-defined handler gains control if there is one; otherwise, the system default handler processes the condition.	Debug does not process this condition. The user-defined handler gains control if there is one; otherwise, the system default handler processes the condition. Debug does not gain control unless a user-defined handler returns with normal status.

Debug Ring

Debug normally runs in the same ring as the program being debugged. You can, however, control the ring in which Debug executes. The SCL command SET_DEBUG_RING specifies the ring in which Debug executes. The Debug ring cannot be set to a ring more privileged than the lowest ring for which you are validated.

You are responsible for ensuring that the program being executed runs in the same ring set for Debug on the SET_DEBUG_RING command. (The ring attributes of the program can be changed using the SCL CHANGE_FILE_ATTRIBUTES command.)

If your program runs entirely in one ring, you need not be concerned with the Debug ring except to understand deferred breaks and multiple breaks.

Deferred Breaks

Breaks that occur in a lower numbered ring than the Debug ring are deferred, or delayed, until execution again reaches the Debug ring. The break is deferred so that you do not get control in a ring more privileged than your own. If you were able to get control at a lower ring, you could view or change data that you normally do not have access to, thereby compromising system security.

Deferred breaks can occur even when your program runs in a single ring. Many of the operating system services used by the program execute in more privileged rings. For example, if you set a read or write break on a status variable used in some NOS/VE request and that variable is accessed in a lower ring, the break is delayed until NOS/VE returns control to your program.

When a break is deferred, Debug issues a special break report message. The break is reported as having happened at the line after the line that made the call, and a second line indicating the actual address of the event is output. The second line is formatted as follows:

Trap deferred from <address>

where address is where the event actually occurred.

Multiple Breaks

Because breaks below the Debug ring are deferred until control returns to the Debug ring, several breaks can be stacked up before Debug gains control. When this happens, Debug must process multiple breaks.

If there are several unprocessed breaks outstanding when Debug gains control, Debug reports each one in the usual way but honors only the first one to occur. No commands are processed for the most recent breaks, not even commands associated with the break definition, since execution of the commands could destroy the environment that existed when the first break occurred.

Multiple breaks can also occur when execution is not below the Debug ring. For example, two terminal breaks or an execution break and a terminal break could occur before Debug gets control. If this ever occurs, Debug honors only the first break.

Multiring Environment

The ability of Debug to function in a multiring environment is limited. If a break event occurs in a lower ring than the Debug ring, Debug gains control, but your options are limited. You can only resume execution of the interrupted procedure or terminate the Debug session. Any program condition handlers established for that event are not processed.

Debug Commands

This section describes the Debug commands in alphabetical order. The Debug commands follow the syntax and conventions for SCL commands, as described in the SCL Language Definition manual. The language elements used as parameters are standard SCL elements as defined in that manual, except for source program names used in the CHANGE_PROGRAM_VALUE and DISPLAY_PROGRAM_VALUE command.

The Debug commands are summarized below.

Command	Description
CHANGE_DEFAULT or **CHANGE_DEFAULTS (CHAD)** *MODULE = name* or *keyword value* *PROCEDURE = name* or *keyword value* *DEBUG_INPUT = file reference* *DEBUG_OUTPUT = file reference* *STATUS = status variable*	Changes the default Debug input/output files and procedure and module names.
CHANGE_MEMORY (CHAM) **ADDRESS = integer** **VALUE = string** or **integer** *TYPE = keyword value* *REPEAT_COUNT = integer* or *keyword value* *STATUS = status variable*	Changes the contents of memory.
CHANGE_PROGRAM_VALUE (CHAPV) **NAME = name** **VALUE = name** *MODULE = name* *PROCEDURE = name* *RECURSION_LEVEL = integer* *RECURSION_DIRECTION = keyword value* *STATUS = status variable*	Changes the value of a program variable.
CHANGE_REGISTER or **CHANGE_REGISTERS (CHAR)** *KIND = keyword value* *NUMBER = keyword value* or *list of integer* **VALUE = integer** or **string** *TYPE = keyword value* *STATUS = status variable*	Changes the contents of the P, A, or X registers.

(Continued)

(Continued)

Command	Description
DELETE_BREAK or **DELETE_BREAKS (DELB)** **BREAK = keyword value** or **list of name** *STATUS = status variable*	Deletes one or more break definitions.
DISPLAY_BREAK or **DISPLAY_BREAKS (DISB)** *BREAK = keyword value* or *list of name* *OUTPUT = file reference* *STATUS = status variable*	Displays specified break definitions.
DISPLAY_CALL or **DISPLAY_CALLS (DISC)** *COUNT = integer* or *keyword value* *START = integer* *OUTPUT = file reference* *STATUS = status variable*	Displays information about the dynamic call chain.
DISPLAY_DEBUGGING_ENVIRONMENT (DISDE) *DISPLAY_OPTIONS = list of keyword value* *OUTPUT = file reference* *STATUS = status variable*	Displays the debugging environment of your session.
DISPLAY_MEMORY (DISM) **address** *BYTE_OFFSET = integer* *BYTE_COUNT = integer* *REPEAT_COUNT = integer* or *keyword value* *OUTPUT = file reference* *STATUS = status variable*	Displays the contents of memory.

(Continued)

(Continued)

Command	Description
DISPLAY_PROGRAM_VALUE (DISPV) **NAME = program name** *MODULE = name* *PROCEDURE = name* *RECURSION_LEVEL = integer* *RECURSION_DIRECTION = keyword value* *OUTPUT = file reference* *STATUS = status variable*	Displays the value of a program value.
DISPLAY_REGISTER or **DISPLAY_REGISTERS (DISR)** *KIND = list of keyword values* *NUMBER = keyword value* or *list of integer* *TYPE = keyword value* *OUTPUT = file reference* *STATUS = status variable*	Displays the contents of the P, A, or X registers.
DISPLAY_STACK_FRAME or **DISPLAY_STACK_FRAMES (DISSF)** *COUNT = integer* or *keyword value* *START = integer* *DISPLAY_OPTION = list of keyword value* *OUTPUT = file reference* *STATUS = status variable*	Displays the contents of one or more stack frames.
QUIT (QUI) *STATUS = status variable*	Terminates the Debug session.
RUN *STATUS = status variable*	Initiates or resumes program execution.

(Continued)

(Continued)

Command	Description
SET_BREAK or **SET_BREAKS (SETB)** **BREAK = name** *EVENT = list of keyword value* **address** *BYTE_OFFSET = integer* *BYTE_COUNT = integer* *COMMAND = string* *STATUS = status variable*	Defines the break.
SET_STEP_MODE (SETSM) **MODE = keyword value** *UNIT = keyword value* *MODULE = keyword value* *PROCEDURE = keyword value* *COMMAND = string* *STATUS = status variable*	Defines a subset of a task to be executed in one step.

CHANGE_DEFAULT (CHAD)

Purpose

Changes the default module, default procedure, default Debug input file, and default Debug output file. The change remains in effect until altered by another CHANGE_DEFAULT command.

Format

CHANGE_DEFAULT or **CHANGE_DEFAULTS**
MODULE=name or *keyword value*
PROCEDURE=name or *keyword value*
DEBUG_INPUT = file reference
DEBUG_OUTPUT = file reference
STATUS = status variable

Parameters

MODULE (M)

Name of the module to be used if the module parameter is not specified in Debug commands that must refer to a module. Specifying the keyword $CURRENT causes the default module to be reset to the module that was executing when Debug gained control.

Omission causes the current default module to remain unchanged.

Debug commands that must refer to a module are:

CHANGE_PROGRAM_VALUE
DISPLAY_PROGRAM_VALUE
SET_BREAK
SET_STEP_MODE

PROCEDURE (P)

Name of the procedure to be used if the procedure parameter is not specified in Debug commands that must refer to a procedure. Specifying the keyword $CURRENT causes the default procedure to be reset to the procedure that was executing when Debug gained control.

Omission causes the current default procedure to remain unchanged.

Debug commands that must refer to a procedure are:

CHANGE_PROGRAM_VALUE
DISPLAY_PROGRAM_VALUE
SET_BREAK
SET_STEP_MODE

DEBUG_INPUT (DI)

File from which Debug commands are read when Debug next gains control. Unless a file position is specified in the file reference, the file is initially positioned at the beginning-of-information; the file is not repositioned in subsequent accesses. Commands are read from the file sequentially. If an end-of-partition or an end-of-file is reached on the input file, program execution resumes.

Omission causes the current Debug input file to remain unchanged. Unless specified otherwise, the initial Debug input file is COMMAND.

DEBUG_OUTPUT (DO)

File on which Debug output is written. The change takes effect immediately. Break report messages and command output are written to this file. Unless a file position is specified in the file reference, the file is initially positioned at the beginning-of-information; the file is repositioned to the beginning-of-information in subsequent accesses.

Omission causes the current Debug output file to remain unchanged. Unless specified otherwise, the initial Debug output file is $OUTPUT.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command specifies that Debug is to read commands from the file DBIN the next time Debug gains control.

```
change_default debug_input = dbin
```

The following command specifies that Debug is to write its output to the file $LIST.

```
change_default debug_output = $list
```

The following command specifies the default module name.

```
change_default module = main
```

CHANGE_MEMORY (CHAM)

Purpose

Changes the contents of memory starting at a specific address. You can change the value of any memory location for which you have write permission.

Format

CHANGE_MEMORY
   ADDRESS = integer
   VALUE = string or **integer**
   TYPE = keyword value
   REPEAT_COUNT = integer or *keyword value*
   STATUS = status variable

Parameters

ADDRESS (A)

Address of the first byte of memory to be changed in the form

rsssoooooooo(16)

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. You can obtain machine addresses by using the cross-reference and load maps for your program.

This parameter is required.

VALUE (V)

New memory value. A string value can be interpreted as a hexadecimal or ASCII string, depending on the value of the TYPE parameter.

A hexadecimal string consists of the hexadecimal digits 0 through 9 and A through F and spaces. Spaces are ignored, but they can be used to improve legibility. Each hexadecimal digit corresponds to 4 bits of memory. The first two digits replace the first byte of memory at the specified address, the second two digits replace the second byte, and so on. If there is an odd number of hexadecimal digits, only the first half of the corresponding byte is changed.

An ASCII string consists of a string of ASCII characters. Each ASCII character corresponds to one byte of memory. The first character replaces the first byte of memory at the specified address, the second character replaces the second byte, and so on.

An integer value completely replaces the contents of eight bytes. A diagnostic message is issued if the integer does not fit into eight bytes.

This parameter is required.

TYPE (T)

Type of data defined by the VALUE parameter. It is one of the following keywords:

ASCII (A)

VALUE is an ASCII string.

HEX (H)

VALUE is a hexadecimal string.

INTEGER (I)

VALUE is an integer.

Omission causes HEX to be used for string values and INTEGER to be used for numeric values.

REPEAT_COUNT (RC)

Number of times VALUE is repeated in memory. It must be a positive integer greater than zero. The address is incremented by the value size each time the value is repeated. The memory change is limited to the end of the data segment containing the specified address. Specifying a value that is too large or specifying the keyword ALL changes all the memory that can be changed.

Omission causes a value of 1 to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

If the CHANGE_MEMORY command contains an error before the STATUS parameter, the remainder of the command is skipped. Therefore, the contents of the STATUS parameter does not reflect the status of the command.

Examples

The following command replaces four bytes of memory beginning at location 0b02200001112 hexadecimal with the hexadecimal string '1010aaab'.

```
change_memory address=0b02200001112(16) ..
  value = '1010aaab'
```

The following command replaces six bytes of memory beginning at location 0b02200000055 hexadecimal with the ASCII string 'STRING'.

```
change_memory address=0b02200000055(16) ..
  value = 'string' type=ascii
```

The following command replaces eight bytes of memory beginning at location 0b02300000223 hexadecimal with the integer value 44.

```
change_memory address = 0b02300000223(16) ..
  value = 44
```

CHANGE_PROGRAM_VALUE (CHAPV)

Purpose

Changes the value of the specified program variable. Replacement values are entered in the same format as defined in your program, not as they are represented in memory.

Format

CHANGE_PROGRAM_VALUE
   NAME = name
   VALUE = name
   MODULE = name
   PROCEDURE = name
   RECURSION_LEVEL = integer
   RECURSION_DIRECTION = keyword value
   STATUS = status variable

Parameters

NAME (N)

Name of the program variable in the source program whose value is to be changed. It can be one of the following:

- Simple variable name.
- Subscripted name.
- Field reference.
- Pointer dereference.

Subscripts can be constants or variables, but not expressions. Substring references are not allowed.

Because names can be long, SCL string variables can be used as aliases for them. To do this, assign a string that contains the identifier to the SCL variable. Then use the SCL variable preceded by a question mark as the value of the NAME parameter.

This parameter is required.

VALUE (V)

New value for the NAME parameter variable. The named VALUE parameter variable must be of the same type as the NAME parameter variable. Combinations allowed for the NAME and VALUE parameters are:

NAME Type	VALUE Type
Integer	Integer constant or variable reference.
Character	Character constant or variable reference.
Boolean	Boolean constant or variable reference.
Ordinal	Ordinal name or variable reference.
Cell	Integer constant or variable reference.
Pointer	Integer constant or variable reference.
String	String constant or variable reference.
Array, record, set, or sequence	Variable reference (byte-aligned and unpacked).

This parameter is required.

MODULE (M)

Name of the module that contains the NAME parameter variable.

Omission causes the module executing when Debug gained control or the module specified by the CHANGE_DEFAULT command to be used.

PROCEDURE (P)

Name of the procedure that contains the NAME parameter variable. If the PROCEDURE parameter is specified, the NAME parameter variable must exist in this procedure or exist in the containing procedure or module. If an inactive procedure is specified, the automatic variables cannot be changed.

Omission causes the procedure executing when Debug gained control or the procedure specified by the CHANGE_DEFAULT command to be used.

RECURSION_LEVEL (RL)

The particular call of a recursive procedure to be used. It must be a positive integer greater than zero. If RECURSION_DIRECTION=FORWARD, a value of 1 is the first call, 2 is the second call (the one called by the first call), and so on. If RECURSION_DIRECTION=BACKWARD, 1 is the most recent call, 2 is the predecessor, and so on.

Recursion only applies to program variables stored on the stack. Recursion cannot apply to variables stored in either a common block or the $STATIC section.

Omission causes a value of 1 to be used.

RECURSION_DIRECTION (RD)

Order in which calls to a recursive procedure are searched. It controls how the value of the RECURSION_LEVEL parameter is interpreted. It can be one of the following keywords:

FORWARD

A RECURSION_LEVEL of 1 specifies that the first call to the procedure is used, a 2 specifies the second call, and so on.

BACKWARD

A RECURSION_LEVEL of 1 specifies that the most recent call to the procedure is used, a 2 specifies its predecessor, and so on.

Recursion only applies to program variables stored on the stack. Recursion cannot apply to variables stored in either a common block or the $STATIC section.

Omission causes BACKWARD to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command changes the value of VARIABLE1.

```
change_program_value name = variable1 value = 3
```

The following command changes the value of INDEX.

```
change_program_value name = index value = 63 ..
  module = ff_pp procedure = gg_pg
```

CHANGE_REGISTER (CHAR)

Purpose Changes the value of the P, A, or X registers that are associated with the procedure executing when Debug gained control.

Format **CHANGE_REGISTER** or **CHANGE_REGISTERS**
KIND = keyword value
NUMBER = keyword value or *list of integer*
VALUE = integer or **string**
TYPE = keyword value
STATUS = status variable

Parameters *KIND (K)*

Register to change. It can be one of the following keywords:

P

The P register.

A

The A registers.

X

The X registers.

Omission causes P to be used.

NUMBER (N)

Which of the register types to change. Its value is a set of one or more integers or ranges of integers from 0 through 15, or the keyword ALL. An informative message is issued for each referenced register whose value was not saved in the current stack frame and, therefore, cannot be changed. This parameter is ignored if KIND=P since there is only one P register.

Omission causes zero to be used.

VALUE (V)

New value of the register. If KIND is P or A, VALUE can be:

- An integer in the range 0 through 0FFFFFFFFFFFF hexadecimal.
- A hexadecimal string containing a maximum of 12 hexadecimal digits (spaces are ignored); each hexadecimal digit corresponds to 4 bits.

The upper 4 bits are ignored when changing the P register since the ring number in P cannot be changed.

If KIND is X, VALUE can be:

- An integer in the range -7FFFFFFFFFFFFFFF hexadecimal through 7FFFFFFFFFFFFFFF hexadecimal.
- A hexadecimal string containing a maximum of 16 hexadecimal digits (spaces are ignored); each hexadecimal digit corresponds to 4 bits.
- An ASCII string containing a maximum of eight ASCII characters; each character corresponds to one byte.

The upper bits of the register are set to zero if an integer is positive or to 1 if an integer is negative and the value does not fill the register. A string value is left-justified with remaining bytes unchanged.

This parameter is required.

TYPE (T)

Type of data specified by the VALUE parameter. It is one of the following keywords:

ASCII (A)

VALUE is an ASCII string.

HEX (H)

VALUE is a hexadecimal string.

INTEGER (I)

VALUE is an integer.

Omission causes HEX to be used for string values and INTEGER to be used for numeric values.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

If the CHANGE_REGISTER command contains an error before the STATUS parameter, the remainder of the command is skipped. Therefore, the contents of the STATUS parameter does not reflect the status of the command.

Examples

The following command changes the current value of the P register to 0A02200004500 hexadecimal. The upper 4 bits for the ring number are ignored.

```
change_register kind = p, ..
  value = 0a02200004500(16)
```

The following command changes the current value of the X7 register to 'ABCDEFGH'.

```
change_register kind = x, number = 7, ..
  value = 'abcdefgh' type = ascii
```

DELETE_BREAK (DELB)

Purpose

Deletes one or more break definitions.

Format

DELETE_BREAK or **DELETE_BREAKS**
  BREAK = keyword value or **list of name**
  STATUS = status variable

Parameters

BREAK or **BREAKS (B)**

Break definitions to be deleted. If the keyword ALL appears in the list of break names, all breaks are deleted. An informative message is issued if a specified break name does not exist; however, all subsequent breaks in the list are processed.

This parameter is required.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

If the DELETE_BREAK command contains an error before the STATUS parameter, the remainder of the command is skipped. Therefore, the contents of the STATUS parameter does not reflect the status of the command.

Examples

The following command deletes break definitions B1, B2, and B3.

```
delete_breaks breaks = (b1,b2,b3)
```

The following command deletes all break definitions.

```
delete_breaks all
```

The following command deletes break definition B4.

```
delete_break b4
```

DISPLAY_BREAK (DISB)

Purpose Displays specified break definitions. The break name, events, address, and any commands associated with the break are displayed.

Format **DISPLAY_BREAK** or **DISPLAY_BREAKS**
BREAK = keyword value or *list of name*
OUTPUT = file reference
STATUS = status variable

Parameters *BREAK* or *BREAKS (B)*

Break definitions to be displayed. If the keyword ALL appears in the list of break names, all break definitions are displayed. An informative message is issued if a specified break name does not exist; however, all subsequent breaks in the list are processed.

Omission causes all break definitions to be displayed.

OUTPUT (O)

File on which the break definitions are written.

Omission causes the current default Debug output file to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples The following command displays break definitions B1, B2, and B5.

```
display_breaks breaks = (b1,b2,b5)
```

The following command displays all break definitions.

```
display_breaks
```

Debug displays the following:

```
-- Break B1
-- event(s) = execution
-- location: M=module_main L=26

-- Break B2
-- event(s) = execution
-- location: M=m L=13 BO=16

-- Break B3
-- event(s) = execution
-- location: M=m L=16

-- Break B4
-- event(s) = execution
-- location: M=perform_integer_multiplications L=7
```

DISPLAY_CALL (DISC)

Purpose Displays information about the dynamic call chain. Usually the procedure name, module name, and line number of each call are shown. Only the procedure or module name and byte offset from the beginning of the procedure or module are shown if you inhibit Debug tables when compiling your program. Only machine addresses are shown for internal NOS/VE calls.

Format **DISPLAY_CALL** or **DISPLAY_CALLS**
COUNT = integer or *keyword value*
START = integer
OUTPUT = file reference
STATUS = status variable

Parameters *COUNT (C)*

Number of calls to be displayed. It must be a positive integer greater than zero or the keyword ALL. If the value specified is greater than the number of existing calls, all calls are displayed.

Omission causes all calls to be displayed.

START (S)

Call on the chain to be displayed first. Thus, it is possible to skip the most recent calls. It must be a positive integer greater than zero. The value 1 represents the most recent call, 2 represents the predecessor of the most recent call, and so forth.

Omission causes the value 1 to be used.

An informative message is issued if the specified number of calls is greater than the actual number of calls.

OUTPUT (O)

File on which the call information is written.

Omission causes the current Debug output file to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command displays the first four calls on the call chain.

```
display_calls count = 4
```

The following command displays all the calls on the call chain beginning with the second most recent call.

```
display_calls start = 2
```

DISPLAY_DEBUGGING_ENVIRONMENT (DISDE)

Purpose

Displays the following information about the environment of your debugging session: current defaults for module, procedure, Debug input file, and Debug output file; the total number of breaks you have set; information about step mode; and the location in your program where execution stopped.

Format

DISPLAY_DEBUGGING_ENVIRONMENT
DISPLAY_OPTIONS = list of keyword value
OUTPUT = file reference
STATUS = status variable

Parameters

DISPLAY_OPTIONS (DO)

Type of information to be displayed. It is one or more of the following keywords:

ALL

Defaults, breaks, step mode attributes, and user addresses are displayed.

BREAKS (B)

The number of breaks you have set, the number of breaks currently in use by Debug, and the number of unused breaks are displayed.

DEFAULTS (D)

The current default values for module, procedure, Debug input file, and Debug output file are displayed.

Unless the CHANGE_DEFAULT command has been specified, the default module and procedure is where execution has stopped in your task. The text $CURRENT is output if module or procedure has not been initialized.

STEP_MODE (SM)

The current step mode attributes are displayed.

USER_ADDRESS (UA)

The location where execution has stopped in your program is displayed.

Omission causes ALL to be used.

OUTPUT (O)

File on which the call information is written. Omission causes the current Debug output file to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command writes defaults, breaks, step mode attributes, and location where execution stopped to the current default Debug output file.

```
display_debugging_environment
```

The following command displays the number of breaks set, the number of unused breaks, and the location where execution stopped.

```
display_debugging_environment ..
  display_options = (b,ua)
```

The following command writes defaults, breaks, step mode attributes, and location where execution stopped to file FILE1 and returns the command status to variable SS.

```
display_debugging_environment ..
  display_options = all output = file1 status = ss
```

DISPLAY_MEMORY (DISM)

Purpose

Displays information located at any address to which you have read access. This command allows you to debug your program even when compiler-generated symbol tables are not available, and to display memory areas that do not correspond to program identifiers. Each display line shows the memory contents in hexadecimal and ASCII formats; the relative byte offset from the initial address is also shown.

The compiler-generated attributes list shows the section name and offset for all variables. You can reference static variables by specifying section name and byte offset. You can reference variables on the stack by specifying the machine address of the stack frame and byte offset. You can obtain the address of the stack frame of the procedure executing when Debug got control by displaying register A1. You can obtain the address of other stack frames by displaying the save area of the wanted stack frame using the DISPLAY_STACK_FRAME command and obtaining the value of register A1 from that stack frame. You can also use the DISPLAY_PROGRAM_VALUE command to display program variables when symbol tables are available.

Format

DISPLAY_MEMORY
address
BYTE_OFFSET = integer
BYTE_COUNT = integer
REPEAT_COUNT = integer or *keyword value*
OUTPUT = file reference
STATUS = status variable

Parameters

address

Memory location to be displayed. The memory location is specified by one or more of the following address parameters:

SECTION = name or keyword value
MODULE = name
ADDRESS = integer

SECTION (SEC)

Memory section containing the data to be displayed. It can be one of the following:

- Working storage section name of a CYBIL program.
- A common block name.
- $BINDING, which is the memory section containing the links to external procedures and the data of the module.
- $LITERAL, which is the memory section containing the literal data (that is, long constants) of the module.
- $STATIC, which is the memory section containing the static (not on the run-time stack) variables not explicitly allocated to a named section of the module.
- CYB$DEFAULT_HEAP, which is the memory section containing the default heap of CYBIL.

When you use SECTION to specify an address, you must qualify it with the MODULE parameter. You can use the BYTE_OFFSET parameter to modify the starting address of memory to be displayed.

Omission indicates that the memory address is specified by the ADDRESS parameter.

MODULE (M)

Module containing the data to be displayed. The MODULE parameter cannot be specified unless the SECTION parameter is specified.

Omission indicates that the memory address is specified by the ADDRESS parameter.

ADDRESS (A)

Address of the first byte of memory to be displayed. Its value is expressed in the form

rsssoooooooo(16)

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. You can use the BYTE_OFFSET parameter to modify the starting address of memory to be displayed.

Omission indicates that the address is specified by the SECTION and MODULE parameters.

BYTE_OFFSET (BO)

Offset to the base address established by one of the address parameters. It must be a positive integer. Its value is added to the base address to form a new address.

The address generated by adding BYTE_OFFSET to the base address must be within the memory block implied by the base address. The block size is the length of the section when the SECTION parameter is specified, and the length of the segment containing the machine address when the ADDRESS parameter is specified.

Omission causes a value of 0 (zero) to be used.

BYTE_COUNT (BC)

Number of bytes in the item to be displayed. It must be a positive integer greater than zero.

Omission causes a value of 1 to be used.

REPEAT_COUNT (RC)

Number of memory areas (items) of length BYTE_COUNT to be displayed. It must be a positive integer. The maximum amount of memory that can be displayed is limited to the block size implied by address (section length for SECTION and segment length for ADDRESS). The keyword ALL or a large integer causes all memory from the specified address to the end of the memory block to be displayed.

Omission causes a value of 1 to be used.

OUTPUT (O)

File on which the displayed information is written.

Omission causes the current Debug output file to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command displays the first three bytes of the literal memory section for module MOD1.

```
display_memory section = $literal module = mod1 ..
  byte_count = 3
```

The following command displays the first four bytes of the memory section DATA1 for module MOD2 as separate items.

```
display_memory section = data1 module = mod2 ..
  repeat_count = 4
```

The following command displays the first 25 bytes of memory starting from the specified address.

```
display_memory address = 0602400000224(16) ..
  byte_count = 8 repeat_count = 25
```

DISPLAY_PROGRAM_VALUE (DISPV)

Purpose Displays the value of the specified program variable (except boolean values) as they appear in the source program, not in hexadecimal format.

Format **DISPLAY_PROGRAM_VALUE**
  NAME = program name
  MODULE = name
  PROCEDURE = name
  RECURSION_LEVEL = integer
  RECURSION_DIRECTION = keyword value
  OUTPUT = file reference
  STATUS = status variable

Parameters **NAME (N)**

Name of the program element whose value is to be displayed. It can be one of the following:

- Simple variable or constant name.
- Subscripted name.
- Field reference.
- Pointer reference.

Subscripts can be constants or variables but not expressions. NAME cannot be a substring.

The variable must be used in your program.

Because names can be long, SCL string variables can be used as aliases for them. To do so, assign the SCL variable to a string containing the identifier. Then use the SCL variable preceded by a question mark as the value of the NAME parameter.

This parameter is required.

MODULE (M)

Name of the module containing the NAME parameter variable.

Omission causes the module executing when Debug gained control or the module specified by the CHANGE_DEFAULT command to be used.

PROCEDURE (P)

Name of the procedure containing the program name. If you specify a procedure that is not in the active call chain, its automatic variables cannot be displayed because it has no stack frame.

Omission causes the procedure executing when Debug gained control to be used if MODULE is also omitted. Otherwise, there is no default procedure when MODULE is specified and PROCEDURE is not; the program name must exist at the module level.

RECURSION_LEVEL (RL)

The particular call of a recursive procedure to be used. It must be a positive integer greater than zero. If RECURSION_DIRECTION=FORWARD, a value of 1 is the first call, 2 is the second call (the one called by the first call), and so on. If RECURSION_DIRECTION=BACKWARD, 1 is the most recent call, 2 is the predecessor, and so on.

Omission causes a value of 1 to be used.

RECURSION_DIRECTION (RD)

Order in which calls to a recursive procedure are searched. It controls how the value of the RECURSION_LEVEL parameter is interpreted. It can be one of the following keywords:

FORWARD (F)

A RECURSION_LEVEL of 1 specifies that the first call to the procedure is used, a 2 specifies the second call, and so on.

BACKWARD (B)

A RECURSION_LEVEL of 1 specifies that the most recent call to the procedure is used, a 2 specifies its predecessor, and so on.

Omission causes BACKWARD to be used.

OUTPUT (O)

File where the display information is written.

Omission causes the current Debug output file to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command displays the value of I from the current module.

```
display_program_value name = i
```

Debug displays the following:

```
i = 879609302207
```

The following command displays the value of DAY from procedure FIRST in the current module.

```
display_program_value name = day ..
  procedure = first
```

DISPLAY_REGISTER (DISR)

Purpose

Displays the contents of the P, A, or X registers that are associated with the procedure executing when Debug gained control.

Format

DISPLAY_REGISTER or **DISPLAY_REGISTERS**
KIND = list of keyword value
NUMBER = keyword value or *list of integer*
TYPE = keyword value
OUTPUT = file reference
STATUS = status variable

Parameters

KIND (K)

Kind of register. It can be one of the following keywords:

P

The P register.

A

The A registers.

X

The X registers.

Omission causes P to be used.

NUMBER (N)

Which of the registers to display. Its value is a set of one or more integers or ranges of integers from 0 through 15, or the keyword ALL. An informative message is issued for each referenced register whose value was not saved in the current stack frame and, therefore, cannot be displayed. This parameter is ignored if KIND=P since there is only one P register.

Omission causes zero to be used.

TYPE (T)

Type of the displayed register values. It can be one of the following keywords:

ASCII (A)

Displays ASCII string values.

HEX (H)

Displays hexadecimal string values.

INTEGER (I)

Displays integer values.

Omission causes HEX to be used for string values and INTEGER for numeric values.

OUTPUT (O)

File on which the register contents are written.

Omission causes the current Debug output file to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command displays the contents of the P register in hexadecimal.

```
display_register p
```

Debugs displays the following:

```
p = b 031 00000040
```

The following command displays the contents of the A8 register in hexadecimal.

```
display_register kind = a number = 8 type = hex
```

The following command displays the contents of the X4, X5, X6, X7, X8, X9, and X10 registers in hexadecimal.

```
display_register kind = x number = 4..10
```

DISPLAY_STACK_FRAME (DISSF)

Purpose Displays the contents of one or more stack frames. Values are displayed in hexadecimal.

Format **DISPLAY_STACK_FRAME** or
DISPLAY_STACK_FRAMES
COUNT = integer or *keyword value*
START = integer
DISPLAY_OPTION = list of keyword value
OUTPUT = file reference
STATUS = status variable

Parameters *COUNT (C)*

Number of stack frames to be displayed. It must be a positive integer. An integer value greater than the number of existing stack frames or the keyword ALL causes all stack frames to be displayed.

Omission causes all stack frames to be displayed.

START (S)

Stack frame to be displayed first. It must be a positive integer greater than zero. The value 1 represents the most recent stack frame, 2 represents its predecessor, and so on.

Omission causes a value of 1 to be used.

DISPLAY_OPTION or *DISPLAY_OPTIONS (DO)*

Area of the stack frames to be displayed. It can be one or more of the following keywords:

AUTO (A)

Area containing the automatic (dynamically allocated) variables of the procedure.

SAVE (S)

Area containing a copy of the registers of the procedure as they existed at the time of a call or trap.

ALL

Both the automatic and save areas.

Omission causes ALL to be used.

OUTPUT (O)

File on which the stack frame values are written.

Omission causes the current Debug output file to be used.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Examples

The following command displays the save area of the most recent stack frame.

```
display_stack_frame display_option = save
```

The following command displays the automatic and save areas of three stack frames beginning with the second most recent one.

```
display_stack_frames count = 3 start = 2
```

The following command displays the automatic and save areas of the most recent stack frame.

```
display_stack_frame  count=1

   STACK FRAME 001        SEGMENT=032
   00000000    00000000  00000000
   00000008    00000000  00000000
   00000010    30300000  00C0FFFF    00
   00000018    80000000  00000000
   00000020    B032B031  00000000     2 1
   00000028    0000B01D  0009B346           F
   00000030    0000B032  00000430       2   0
   00000038    0040B032  00000400     @ 2
   00000040    FF77B032  000003C0     w 2
   00000048    FFFCB01B  00020F78           x
   00000050    0000B032  00000390       2

   SAVE AREA

   P=B 035 00000026        VMID=0
   UM=FFF7    UCR=0040    MCR=0000

   A0=B 032 00000460      A1=B 032 00000408
   A2=B 032 000003C0      A3=B 030 00000000
   A4=B 032 00000390      A5=B 02F 00000020
   A6=B 02E 00000000      A7=B 02F 00000000
   A8=B 00F 00000018      A9=B 032 00000630
   AA=B 032 00000A30      AB=F FFF 80000000
   AC=F FFF 80000000      AD=B 032 00001058
   AE=F FFF 80000000      AF=B 00B 000557F8

   X0=0000B01D  00020060      X1=00000000  00000000
   X2=0000FFFF  80000000      X3=000007FF  FFFFFFFF

   X4=00000000  10000000      X5=00000000  00000008
   X6=00000000  0000000D      X7=00000000  0000001D
   X8=00000000  00000000      X9=00000000  00000008
   XA=00000000  00000300      XB=00000000  00000000
   XC=00000000  00000001      XD=00000000  00000022
   XE=00000000  00010040      XF=00000000  0000004E
```

QUIT (QUI)

Purpose Terminates the Debug session and returns control to the NOS/VE operating system. The session is terminated immediately; the program is not executed to completion.

Format **QUIT**
STATUS = *status variable*

Parameter *STATUS*

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

RUN

Purpose

Initiates or resumes program execution once Debug has gained control. Execution continues until Debug again gains control. If the program has run to completion, entering the RUN command terminates the program and returns control to the NOS/VE operating system.

Execution begins at the instruction whose address is contained in the P register or at the condition handler (if there is one) of the program for the event that caused Debug to gain control. (Refer to the SCL Language Definition manual for a discussion of condition handlers.) If the P register points to the instruction that caused the event (such as division by zero), the same event will occur immediately after entering the RUN command. In this case, you must change the value in the P register (use the CHANGE_REGISTER command) or change the value of one of the operands (use the CHANGE_PROGRAM_VALUE command) before entering the RUN command.

When Debug processes the RUN command, all previously created SCL blocks (except SET_BREAK command information and the name of the current Debug input file) are lost. This means that all information about SCL commands, such as IF/THEN blocks or WHILE/FOR loops that span RUN commands are lost. Global variables must be recreated with XREF. For example, the following sequence recreates the variable COUNT:

```
DB/create_variable name = count ..
DB../kind = integer scope = job value = 0
DB/set_break break = one line = one ..
DB../run  "BREAK ONE"
DB/create_variable name = count ..
DB../kind = integer scope = xref
```

Format

RUN
  STATUS = *status variable*

Parameter

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

SET_BREAK (SETB)

Purpose

Defines the break. You can specify one or more events and the location at which Debug is to take control. When the specified event occurs, program execution is suspended and a message informs you which break occurred. At this point, you can enter another Debug command, or any command that can be processed by the operating system or an active command utility.

Format

SET_BREAK or **SET_BREAKS**
  BREAK = name
  EVENT = list of keyword value
  address
  BYTE_OFFSET = integer
  BYTE_COUNT = integer
  COMMAND = string
  STATUS = status variable

Parameters

BREAK (B)

Name of the break definition. This name is used to reference the break definition in the DISPLAY_BREAK and DELETE_BREAK commands. This name is displayed in the break report message when the break occurs. A break cannot be named ALL because ALL is used as a keyword in other Debug commands.

This parameter is required.

EVENT or *EVENTS (E)*

Events that must occur for the break to occur. If you specify more than one event, the break is honored if only one of the events occurs. Possible events can be any of the following keywords:

ARITHMETIC_OVERFLOW (AO)

Break when an arithmetic overflow occurs on an instruction in the specified address range. The P register points to the instruction that caused the overflow.

ARITHMETIC_SIGNIFICANCE (AS)

Break when arithmetic significance is lost on an instruction in the specified address range. The P register points to the instruction that caused the loss of significance.

BRANCH (B)

Break before a branch to or a return from any location in the specified address range occurs.

CALL (C)

Break before a subprogram call occurs to any address in the specified address range.

DIVIDE_FAULT (DF)

Break when division by zero occurs in an instruction in the specified address range. The P register points to the instruction that caused the division by zero.

EXECUTION (E)

Break before the instruction in the specified address range is executed.

If the address is specified by the line number, not every line is usable. For example, breaks cannot be set at IFEND statements because it is not obvious when contol reaches them.

EXPONENT_OVERFLOW (EO)

Break when an exponent overflow occurs in an instruction in the specified address range. The P register points to the instruction following the one that caused the overflow.

EXPONENT_UNDERFLOW (EU)

Break when an exponent underflow occurs in an instruction in the specified address range. The P register points to the instruction following the one that caused the underflow.

FLOATING_POINT_INDEFINITE (FPI)

Break when the result of a floating-point operation is indefinite in an instruction in the specified address range. The P register points to the instruction following the one that caused the results to be indefinite.

FLOATING_POINT_SIGNIFICANCE (FPS)

Break when significance is lost during a floating-point operation in an instruction in the specified address range. The P register points to the instruction following the one that caused the loss of significance. This event will not occur unless your program sets the floating-point loss-of-significance bit in the user mask register.

INVALID_BDP_DATA (IBD)

Break when a business data processing (BDP) instruction fault occurs in an instruction in the specified address range. The P register points to the instruction that caused the fault. The BDP instructions are described in volume II of the virtual state hardware reference manual.

READ (R)

Break before a read occurs from the specified address range. The break occurs only if the first byte of the item to be read is within the address range.

READ_NEXT_INSTRUCTION (RNI)

Break before the instruction in the specified address range is executed.

WRITE (W)

Break before a write occurs into the specified address range. The break occurs only if the first byte of the item to be written is within the address range.

Omission causes EXECUTION to be used.

Debug gains control when the following events occur even if you do not set a break for them:

ARITHMETIC_OVERFLOW
ARITHMETIC_SIGNIFICANCE
DIVIDE_FAULT
EXPONENT_OVERFLOW
EXPONENT_UNDERFLOW
FLOATING_POINT_INDEFINITE
FLOATING_POINT_SIGNIFICANCE
INVALID_BDP_DATA

Specific breaks can be set for these events, however, so that a predefined set of commands can be executed when Debug gains control.

address

Location at which the break occurs. For the break to occur, the specified event must occur within the range defined by the address parameters. All address parameters are interpreted as a single address. You can use the BYTE_COUNT and BYTE_OFFSET parameters to specify an address range. The address parameters are:

LINE = integer
SECTION = name or keyword value
MODULE = name
PROCEDURE = name
ENTRY_POINT = name
ADDRESS = integer

LINE (L)

Line at which Debug gains control. Unless the MODULE parameter is also specified, the line number must exist in the module that was executing when Debug gained control or the default module set with the CHANGE_DEFAULT command.

You can use BYTE_OFFSET and BYTE_COUNT to modify this parameter.

Not all lines of a program can be referenced. Only executable statements that begin on a separate line can be referenced. A second or third statement on a line or a line containing the continuation of a statement cannot be referenced. In addition, IFEND lines cannot be referenced.

Omission indicates that the break address is specified by another parameter.

SECTION (SEC)

A memory section. It can be one of the following:

- Name of the working storage section as declared in the source program.
- Name of a common block.
- $BINDING, which is the memory section containing the links to external procedures and the data of the module.
- CYB$DEFAULT_HEAP, which is the memory section containing the default heap for CYBIL.
- $LITERAL, which is the memory section containing the literal data (that is, long constants) of the module.
- $STATIC, which is the memory section containing the static (not on the run-time stack) variables that are not allocated to an explicitly named section of the module.

Unless the MODULE parameter is also specified, the section must exist for the module that was executing when Debug gained control or the default module set with the CHANGE_DEFAULT command. The SECTION parameter cannot be specified for modules that are components of a bound module unless the section is a common block (refer to the discussion under Addressing Bound Modules earlier in this chapter). You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify this parameter.

Omission indicates that the break address is specified by another parameter.

MODULE (M)

An address or qualification of another address specifier. If used alone, the MODULE parameter specifies an address (the first byte of the first code section of the module). Module represents only the first code section. MODULE cannot reference the code section of a component module of a bound module (refer to the discussion under Addressing Bound Modules earlier in this chapter). If used with the LINE, SECTION, or PROCEDURE address parameters, the MODULE parameter identifies the module containing the line, section, or procedure. If used to specify an address, the BYTE_OFFSET and BYTE_COUNT parameters can be used to modify the MODULE parameter.

Omission causes the module executing when Debug gained control or the default module set with the CHANGE_ DEFAULT command to be used.

PROCEDURE (P)

An address (the first byte of the code section of the procedure). Unless the MODULE parameter is also specified, the procedure must exist in the module that was executing when Debug gained control or the default module set with the CHANGE_DEFAULT command. You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify the PROCEDURE parameter. You cannot specify the LINE or SECTION address parameters with the PROCEDURE parameter.

When a name is specified, it indicates the procedure to be used. The name must be a procedure, function, or program.

Omission indicates that the break address is specified by another parameter.

ENTRY_POINT (EP)

An entry point expressed as a name known to the loader. It must be a procedure or data name with an XDCL attribute subject to certain restrictions. (Refer to Attributes in chapter 3 for a description of the XDCL attribute. Also, refer to the SCL Object Code Management manual for further information on restrictions.) You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify the ENTRY_POINT parameter. You cannot use other address parameters with this parameter.

Omission indicates that the break address is specified by another parameter.

ADDRESS (A)

Address of the break event in the form

rsssoooooooo(16)

where r is the ring number, sss is the segment number, and oooooooo is the offset within the segment. You can obtain machine addresses from the cross-reference and load maps for your program. You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify the ADDRESS parameter. You cannot use other address parameters with this parameter.

Omission indicates that the break address is specified by another parameter.

The address parameter is required.

BYTE_OFFSET (BO)

Offset to the base address established by one of the address parameters. It must be a positive integer. Its value is added to the base address to form a new address. The break is then set for this new address.

Omission causes a value of zero to be used.

BYTE_COUNT (BC)

Number of bytes in an address range. It must be a positive integer greater than zero.

Omission causes a value of 1 to be used.

COMMAND or *COMMANDS (C)*

String of commands to be executed when the break is honored. These commands can be processed by Debug, the operating system, or other active command processor. If a command in the string includes a quoted string, that string must be enclosed in two single quotes. After the commands in the string have been executed, commands are read from the current Debug input file unless the string contains a RUN command.

No break report message is issued before the commands in the string are executed. If you want a message to be displayed, include an SCL DISPLAY_VALUE command in the string.

If an error is detected in one of the commands in the string, the break report message is issued, the error is reported, and commands are read from the Debug input file. The remaining commands in the string are not executed.

Omission indicates that no commands are associated with the break. Commands are read from the Debug input file.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

If the SET_BREAK command contains an error before the STATUS parameter, the remainder of the command is skipped. Therefore, the contents of the STATUS parameter does not reflect the status of the command.

Examples

The following command causes a break when execution reaches line 10 of module PROG1.

```
set_break break = b1 line = 10 module = prog1
```

The following command causes a break when a branch or return occurs to line 40 of the module executing when Debug gained control.

```
set_break break = b2 event = branch line = 40
```

SET_STEP_MODE (SETSM)

Purpose

Enables you to execute a specified subset of a task and receive control.

If step mode is activated, a RUN command causes your program to execute for the specified unit. You are then prompted for further command input. A string of commands can be associated with the step and will be processed each time the step is completed. Stepping with a unit of line or procedure is only available if the source program was compiled with the Debug optimization option (OPTIMIZATION=DEBUG).

Activating step mode is an effective debugging aid but is expensive in terms of execution time.

Format

SET_STEP_MODE
MODE = keyword value
UNIT = keyword value
MODULE = keyword value
PROCEDURE = keyword value
COMMAND = string
STATUS = status variable

Parameters

MODE

Activates or deactivates step mode. It can be one of the following keywords:

ON

Activates step mode. When step mode is on, a RUN command causes one step to be executed. A step is defined by the UNIT parameter.

OFF

Deactivates step mode. When step mode is off, any remaining parameters are ignored.

If you specify ON and step mode is already on, all previous values are replaced with the new values.

This parameter is required.

UNIT (U)

Length of the step. It can be one of the following keywords:

PROCEDURE (P)

The step is reported each time a new procedure begins and after any prologue code for the procedure has executed.

LINE (L)

The step is reported before the code is executed for each line except for the procedure lines.

Omission causes LINE to be used.

MODULE (M)

The modules reported. This parameter is used with the UNIT parameter. It can be one of the following keywords:

$ALL

Reports a step that is in any module.

$CURRENT

Reports a step only if the step occurs in the module where the program is executing when step mode is activated.

You cannot specify both the MODULE and the PROCEDURE parameters in the same SET_STEP_MODE command.

Omission causes the current value for the default module to be used.

PROCEDURE (P)

The procedures reported. This parameter is used with the UNIT parameter. It can be one of the following keywords:

$ALL

Reports a step that is in any procedure.

$CURRENT

Reports a step only if the step occurs in the procedure where the program is executing when step mode is activated.

You cannot specify both the PROCEDURE and MODULE parameters in the same SET_STEP_MODE command.

Omission causes $ALL to be used.

COMMAND or *COMMANDS (C)*

String of commands that will be executed when the step occurs. If the command string includes a RUN command, the task is resumed and the step is not reported. If the string does not include a RUN command, command input is requested from the current Debug input file.

STATUS

Return status of the command. If omitted and an error does not occur, Debug processes the next command. If omitted and an error occurs, the status value is returned to $RESPONSE and to the Debug output file if $RESPONSE is connected to that file. This file is normally connected during interactive debugging.

Example

The following commands activate step mode with a unit of line in the current module, execute the entire program automatically, display each line executed, and then deactivate step mode.

```
set_step_mode mode = on ..
  command = 'display_debugging_environment ..
  display_options = ua; RUN' ..
run
set_step_mode mode = off
quit
```

Debug Functions

Debug functions are intended for use with SCL during a Debug session. These functions are only available while Debug has control. They are not known when your program is executing or after the Debug session has been terminated.

$CURRENT_LINE

Purpose Returns the current line number value from the program at the point where Debug has control.

Format **$CURRENT_LINE**

Example

```
if $current_line < 100 then
   display_calls
ifend
```

$CURRENT_MODULE

Purpose Returns the name of the module where execution is stopped.

Format **$CURRENT_MODULE**

Example

```
if $current_module = main then
  set_break name = break1 line = 234
ifend
```

$CURRENT_PROCEDURE

Purpose Returns the name of the procedure where execution is stopped.

Format **$CURRENT_PROCEDURE**

Example

```
set_step_mode mode = on unit = procedure ..
  command ='if $current_procedure = sub2 then; ..
set_step_mode mode = on unit = line; ..
  else; run; ifend'
```

$CURRENT_PVA

Purpose Returns an integer value for the process virtual address (PVA) where execution is stopped.

Format **$CURRENT_PVA**

Example

```
if $current_pva > 0b03500000026(16) then
  display_calls
ifend
```

$PROGRAM_VALUE

Purpose

Returns the value of a program value (specified as the name positional value). Input positional values for module, procedure, recursion_level, and recursion_direction can be specified to fully identify the named variable.

The $PROGRAM_VALUE function allows you to incorporate the values of program variables in SCL statements in order to enhance debugging capabilities both in interactive and batch sessions.

Format

$PROGRAM_VALUE(name,*module,procedure, recursion_level,recursion_direction***)**

Parameters

name

Name of the program element whose value is to be displayed. It can be one of the following:

- Simple variable.
- Subscripted name.
- Field reference.
- Pointer reference.

The named variable must be used in your program.

Because names can be long, SCL string variables can be used as aliases for them. To do this, assign the SCL variable to a string containing the identifier. Then use the SCL variable preceded by a question mark as the value of the name parameter.

This parameter is required.

module

Name of the module that contains the name parameter variable. Omission causes the module executing when Debug gained control or the module specified by the CHANGE_ DEFAULT command to be used.

procedure

Name of the procedure that contains the name parameter variable. If you specify a procedure that is not in the active call chain, its automatic variables cannot be used because it has no stack frame. Omission causes the procedure executing when Debug gained control to be used if a module name is not specified. Otherwise, there is no default procedure when a module name is specified and a procedure name is not specified; the name parameter variable must exist at the module level.

recursion_level

The particular call of a recursive procedure to be used. It must be a positive integer greater than zero. If recursion_ direction=FORWARD, a value of 1 is the first call, 2 is the second call (the one called by the first call), and so on. If recursion_direction=BACKWARD, 1 is the most recent call, 2 is the predecessor, and so on.

Omission causes a value of 1 to be used.

recursion_direction

Order in which calls to a recursive procedure are searched. It controls how the value of the recursion_level parameter is interpreted. It can be one of the following keywords:

FORWARD (F)

A recursion_level of 1 specifies that the first call to the procedure is used, a 2 specifies the second call, and so on.

BACKWARD (B)

A recursion_level of 1 specifies that the most recent call to the procedure is used, a 2 specifies its predecessor, and so on.

Omission causes BACKWARD to be used.

Example

```
set_break name = b1 line = 23 command = ..
  'if $program_value(index) < 45 then; run; ifend'
```

Using Debug

This section illustrates the use of Debug. Major Debug commands are illustrated in the sample Debug sessions.

Sample Debug Sessions

Debug can be used in interactive or batch mode. Two Debug sessions follow. The first session illustrates using Debug interactively. The second session illustrates using Debug in batch mode.

Interactive Debug Session

The source listing of the CYBIL program used in this interactive Debug session is shown in figure 9-1.

```
SOURCE LIST OF module_main

       0    1 MODULE module_main;
       0    2
       0    3   PROCEDURE [XREF] p (operand1,
       0    4         operand2: integer;
       0    5     VAR result: integer;
       0    6     VAR status: boolean);
       0    7
       0    8   PROGRAM main;
       0    9
       0   10     VAR
       4   11       i,
       4   12       j,
       4   13       k: [STATIC] integer,
       4   14       x,
       4   15       y,
       4   16       z: integer,
       4   17       b: boolean;
       4   18
       4   19     i := 7fffffffffffff(16);
      14   20     j := 10000000(16);
      1C   21     k := i * j;
      2A   22     k := i DIV j;
      38   23     FOR x := 0 TO 100 DO
      42   24       y := x * x - 500;
      54   25       p (x, y, z, b);
      80   26       IF b THEN
      88   27         EXIT main;
      8A   28       IFEND;
      8A   29     FOREND;
      8E   30   PROCEND main;
       0   31 MODEND module_main;
```

Figure 9-1. Source Listing for Interactive Debug Session

(Continued)

(Continued)

```
SOURCE LIST OF m

       0     1 MODULE m;
       0     2
       0     3   PROCEDURE [XDCL] p (operand1,
       0     4         operand2: integer;
       0     5     VAR result: integer;
       0     6     VAR b: boolean);
       0     7
       0     8     PROCEDURE [XREF] mult (a,
       4     9           b: integer;
       4    10       VAR c: integer);
       4    11
       4    12     IF operand1 < operand2 THEN
      10    13       mult (operand1, operand2, result);
      34    14       b := TRUE;
      3E    15     ELSE
      42    16       b := FALSE;
      4C    17     IFEND;
      4C    18   PROCEND p;
       0    19 MODEND m;


SOURCE LIST OF perform_integer_multiplications

       0     1 MODULE perform_integer_multiplications;
       0     2
       0     3   PROCEDURE [XDCL] mult (a,
       0     4         b: integer;
       0     5     VAR c: integer);
       0     6
       0     7     c := a * b;
      16     8   PROCEND mult;
       0     9 MODEND perform_integer_multiplications;
```

Figure 9-1. Source Listing for Interactive Debug Session

The following command compiles the CYBIL program:

```
cybil i=sample, l=list, b=lgo
```

The name of the file containing the object code of the program is LGO. The following command initiates a Debug session:

```
execute_task file=lgo debug_mode=on
```

Debug issues a banner and the Debug prompt, DB/, indicating that Debug has control. Entering the RUN command initiates program execution:

```
DEBUG
DB/ run
--  DEBUG: arithmetic_overflow at M=module_main L=21 B0=8
DB/
```

By looking at the source listing for MODULE_MAIN, line 21, you can see that the overflow occurred during a multiplication operation. Entering the following commands allows you to view the values of the variables I and J:

```
DB/ display_program_value name=i
i =  281474976710655
DB/ display_program_value name=j
j =  268435456
```

When I and J are multiplied, the result exceeds the maximun value allowed; therefore, arithmetic overflow occurs. Since the P register points to the instruction that caused the overflow, entering the RUN command would cause the overflow to reoccur. Changing the P register allows program execution to continue. The following commands accomplish this:

```
DB/ display_register kind=p
P=B 035 00000024
DB/ change_register kind=p value=0b03500000026(16)
```

Since the value in the P register begins with a letter, a leading zero is required for the value parameter. Because the value parameter is in hexadecimal, the radix is required. The following command shows that the P register is indeed changed:

```
DB/ display_register kind=p
P=B 035 00000026
```

Setting the following breaks allows you to follow program execution:

```
DB/ set_break break=prog_main module=module_main line=26
DB/ set_break break=proc_p1 module=m procedure=p ..
DB../byte_offset=34(16)
DB/ set_break break=proc_p2 line=16 module=m
DB/ set_break break=proc_mult line=7 ..
DB../module=perform_integer_multiplications
```

The first break set, PROG_MAIN, does not require the module/procedure parameters because it is for the module/procedure executing when Debug gained control. The address for the second break set, PROC_P1, is specified in terms of module/procedure offset addressing; the hexadecimal offset is obtained from the first column of numbers on the source listing. (Since line numbers are available, the break address is reported as a line number.) The third and fourth breaks set, PROC_P2 and PROC_MULT, require the module and procedure parameters since they are not set for the current module/procedure.

Entering the RUN command causes the program to execute until the first break is reached. The DISPLAY_CALLS command allows you to trace program execution:

```
DB/ run
 --  DEBUG: break PROC_P2, execution at M=m L=16
DB/ display_calls
 --  Traceback from procedure P module M at line 16
 --  Called from procedure MAIN module MODULE_MAIN at line 25
        byte offset 40
DB/ run
 --  DEBUG: break PROG_MAIN, execution at M=module_main L=26
DB/ display_calls
 --  Traceback from procedure MAIN module MODULE_MAIN at
        line 26
```

At this point, you could enter any other command. For example, you could enter the DISPLAY_STACK_FRAME command and then the RUN command:

```
DB/ display_stack_frame
STACK FRAME 001          SEGMENT=036
00000000     00000000  00000000
00000008     00000000  00000000
00000010     0000B035  00000000       5
00000018     00000000  00000000
00000020     FFFFFFFF  FFFFFE0C
00000028     0000B036  00000398       6
00000030     00000000  00000000
00000038     00000000  00000000
00000040     FFFFFFFF  FFFFFE0C
00000048     B0360000  04084834     6    H4
00000050     B0360000  04170428     6     (

SAVE AREA

P=B 035 00000080          VMID=0
UM=FFF7    UCR=0080      MCR=0000

A0=B 036 00000438        A1=B 036 000003E0
A2=B 036 00000398        A3=B 034 00000000
A4=B 036 00000370        A5=B 036 00000417
A6=B 036 00000417        A7=B 036 00000250
A8=B 034 00000080        A9=B 011 00000408
AA=B 011 00000168        AB=B 011 00000608
AC=B 00B 00021A10        AD=B 006 00002C40
AE=F FFF 80000000        AF=B 036 00000398

X0=0000B035  00040243       X1=0000B036  00000398
X2=00000000  00000000       X3=00000000  00000064
X4=FFFFFFFF  FFFFFE0C       X5=00000000  00000000
X6=00000000  0000000F       X7=00000000  00989680
X8=00000000  00000022       X9=00000000  00000012
XA=00000000  0000004E       XB=00000000  00000000
XC=00000000  0000015E       XD=00000000  00000003
XE=00000000  00000751       XF=00000000  00000001
DB/ run
 --  DEBUG: break PROC_P2, execution at M=m L=16
```

Since CYBIL variable names can be long, you can assign an SCL variable to that name and then use the SCL variable prefixed by ? in a Debug command. For example, the variable OPERAND1 in procedure P can be shortened to OP1 as follows:

```
DB/ op1='operand1'
DB/ display_program_value name=?op1
operand1 =  1
```

By looking at the source listing, you can see that the program is in a loop that executes 100 times. To avoid encountering the two breaks that are in the loop, you can delete them. First, you can display the break definitions to obtain the break names. Then delete them. Displaying the breaks again shows that two breaks were eliminated:

```
DB/ display_breaks

 --  Break PROG_MAIN
 --  event(s) = execution
 --  location: M=module_main L=26

 --  Break PROC_P1
 --  event(s) = execution
 --  location: M=m L=14

 --  Break PROC_P2
 --  event(s) = execution
 --  location: M=m L=16

 --  Break PROC_MULT
 --  event(s) = execution
 --  location: M=perform_integer_multiplications L=7

DB/ delete_breaks break=(prog_main,proc_p2)
DB/ display_breaks

 --  Break PROC_P1
 --  event(s) = execution
 --  location: M=m L=14

 --  Break PROC_MULT
 --  event(s) = execution
 --  location: M=perform_integer_multiplications L=7
```

Instead of entering two DELETE_BREAK commands, both breaks are specified in the same DELETE_BREAK command.

When displaying the call chain, you can skip one or more of the most recent calls. Note the difference in address reporting when the first call is skipped:

```
DB/ run
 --  DEBUG: break PROC_MULT, execution at
M=perform_integer_multiplications L=7
DB/ display_calls
 --  Traceback from procedure MULT module
PERFORM_INTEGER_MULTIPLICATIONS at line 7
 --  Called from procedure P module M at
        line 13 byte offset  BC=10(16)
 --  Called from procedure MAIN module MODULE_MAIN at line 25
byte offset 40
DB/ display_calls start=2
 --  Traceback from procedure P module M at line 13
 --  Called from procedure MAIN module MODULE_MAIN at line 25
byte offset 40
```

Displaying program names is an important aspect of debugging. You can display the value of program names in other procedures/modules as well as in the current ones. All static names can be displayed. Only automatic names in the active call chain, however, can be displayed. To obtain the active call chain, enter the DISPLAY_CALLS command:

```
DB/ run
 --  DEBUG: break PROC_P1, execution at M=m L=14
DB/ display_calls
 --  Traceback from procedure P module M at line 14
 --  Called from procedure MAIN module MODULE_MAIN at line 25
        byte offset 40
```

Procedures MAIN and P are active. You can, therefore, display the value of any variable within these procedures. To display the value of B in procedure P, enter the following command:

```
DB/ display_program_value name=b module=module_main ..
..DB/procedure=main
b =  FALSE
```

To display the value of B in procedure MAIN, enter the following command:

```
DB/ display_program_value name=b module=module_main procedure=main
b =  FALSE
```

Since module MODULE_MAIN and procedure MAIN are not the current module and procedure, the MODULE and PROCEDURE parameters are required.

Entering the RUN command one more time causes the program to terminate. To terminate the Debug session, enter the QUIT command:

```
DB/ run
 --  DEBUG: program terminated by returning
DB/ quit
 --  DEBUG: QUIT terminated task
```

At this point, the operating system prompt, /, appears and you can enter any SCL command.

Batch Debug Session

The source listing of the CYBIL program used in this batch Debug session is shown in figure 9-2. The name of the file containing the object code of the program is SAMPLE2. SAMPLE2 is essentially the same as the program used in the interactive session. The command stream used for the batch session is shown in figure 9-3. The Debug commands used are the same as those used in the interactive session.

```
SOURCE LIST OF module_main

     0      1 MODULE module_main;
     0      2
     0      3   PROCEDURE [XREF] p (operand1,
     0      4         operand2: integer;
     0      5     VAR result: integer;
     0      6     VAR status: boolean);
     0      7
     0      8   PROGRAM main;
     0      9
     0     10     VAR
     4     11       x,
     4     12       y,
     4     13       z: integer,
     4     14       b: boolean;
     4     15
     4     16     FOR x := 0 TO 100 DO
     E     17       y := x * x - 500;
    20     18       p (x, y, z, b);
    4C     19       IF b THEN
    54     20         EXIT main;
    56     21       IFEND;
    56     22     FOREND;
    5A     23   PROCEND main;
     0     24 MODEND module_main;
```

Figure 9-2. Source Listing for Batch Debug Session

(Continued)

(Continued)

```
SOURCE LIST OF m

    0     1 MODULE m;
    0     2
    0     3   PROCEDURE [XDCL] p (operand1,
    0     4         operand2: integer;
    0     5     VAR result: integer;
    0     6     VAR b: boolean);
    0     7
    0     8     PROCEDURE [XREF] mult (a,
    4     9           b: integer;
    4    10       VAR c: integer);
    4    11
    4    12     IF operand1 < operand2 THEN
   10    13       mult (operand1, operand2, result);
   34    14       b := TRUE;
   3E    15     ELSE
   42    16       b := FALSE;
   4C    17     IFEND;
   4C    18   PROCEND p;
    0    19 MODEND m;

SOURCE LIST OF perform_integer_multiplications

    0     1 MODULE perform_integer_multiplications;
    0     2
    0     3   PROCEDURE [XDCL] mult (a,
    0     4         b: integer;
    0     5     VAR c: integer);
    0     6
    0     7     c := a * b;
   16     8   PROCEND mult;
    0     9 MODEND perform_integer_multiplications;
```

Figure 9-2. Source Listing for Batch Debug Session

The following numbered paragraphs correspond to the numbers in figure 9-3.

① The COLLECT_TEXT command collects the Debug commands on the file named BATCH_SESSION. All commands are placed on BATCH_SESSION until the double asterisks are encountered.

② The CREATE_VARIABLE command creates an SCL variable of type STATUS to be used on the Debug SET_BREAK commands.

③ The IF/IFEND command is used to check the status variable. If the status variable is not provided and the command is in error, the session will be terminated.

④ Indicates that commands are no longer collected on the file BATCH_SESSION.

⑤ The four CREATE_FILE_CONNECTION commands cause a complete record of the Debug session to be recorded on file SESSION.

⑥ The EXECUTE_TASK command initiates the Debug session. Notice that the Debug input file is BATCH_SESSION.

⑦ The standard file $OUTPUT must be disconnected from SESSION before that file can be copied.

⑧ The COPY_FILE command causes the file SESSION to be copied to file $OUTPUT, which is printed at the end of the job. The contents of this file are shown in figure 9-4. Notice that the Debug prompt, DB/, is replaced by CI or CS because of the file connections.

```
   login family_name=... user=... password=... job_class=batch
①  collect_text output=batch_session
②  create_variable name=stat kind=status
   set_break break=prog_main line=19 module=module_main status=stat
③  if stat.normal = false then;
   display_value 'break prog_main failed'
   ifend
   set_break break=proc_p1 line=14 module=m status=stat
   if stat.normal = false then; display_value 'break proc_p1 failed'
   ifend
   set_break break=proc_p2 line=16 module=m status=stat
   if stat.normal = false then; display_value 'break proc_p2 failed'
   ifend
   set_break break=proc_mult line=7 ..
   module=perform_integer_multiplications status=stat
   if stat.normal = false then;
   display_value 'break proc_mult failed'
   ifend
   run
   display_calls
   run
   display_calls
   display_stack_frame
   run
   display_breaks
   delete_breaks break=(prog_main,proc_p2)
   display_breaks
   run
   display_calls
   display_calls start=2
   run
   display_calls
   display_program_value name=b
   display_program_value name=b module=module_main procedure=main
   run
   quit
④  **
   attach_file file=sample2
⑤  create_file_connection $output session
   create_file_connection $response session
   create_file_connection $errors session
   create_file_connection $echo session
⑥  execute_task file=sample2 debug_input=batch_session ..
   debug_output=session debug_mode=on
⑦  delete_file_connection $output session
⑧  copy_file session
   logout
```

Figure 9-3. Command Stream for Batch Debug Session

```
 CI execute_task file=sample2 debug_input=batch_session
debug_output=session debug_mode=on
DEBUG
 CI create_variable name=stat kind=status
 CI set_break break=prog_main line=19 module=module_main status=stat
 CI if stat.normal = false then
 CS  display_value 'break prog_main failed'
 CS ifend
 CI set_break break=proc_p1 line=14 module=m status=stat
 CI if stat.normal = false then
 CS  display_value 'break proc_p1 failed'
 CS ifend
 CI set_break break=proc_p2 line=16 module=m status=stat
 CI if stat.normal = false then
 CS  display_value 'break proc_p2 failed'
 CS ifend
 CI set_break break=proc_mult line=7
 module=perform_integer_multiplications status=stat
 CI if stat.normal = false then
 CS  display_value 'break proc_mult failed'
 CS ifend
 CI run
 --  DEBUG: break PROC_P2, execution at M=m L=16
 CI display_calls
 --  Traceback from procedure P module M at line 16
 --  Called from procedure MAIN module MODULE_MAIN at line 18 byte
 offset 40
 CI run
 --  DEBUG: break PROG_MAIN, execution at M=module_main L=19
 CI display_calls
 --  Traceback from procedure MAIN module MODULE_MAIN at line 19
```

Figure 9-4. Batch Debug Session

(Continued)

(Continued)

```
 CI display_stack_frame
STACK FRAME 001        SEGMENT=035
00000000    00000000  00000000
00000008    00000000  00000000
00000010    0000B034  00000000        4
00000018    00000000  00000000
00000020    FFFFFFFF  FFFFFE0C
00000028    0000B035  00000398        5
00000030    00000000  00000000
00000038    00000000  00000000
00000040    FFFFFFFF  FFFFFE0C
00000048    B0350000  04084834      5    H4
00000050    B0350000  04170428      5     (

SAVE AREA

P=B 034 0000004C         VMID=0
UM=FFF7    UCR=0080     MCR=0000

A0=B 035 00000438       A1=B 035 000003E0
A2=B 035 00000398       A3=B 033 00000000
A4=B 035 00000370       A5=B 035 00000417
A6=B 035 00000417       A7=B 035 00000250
A8=B 033 00000080       A9=B 011 00000408
AA=B 011 00000168       AB=B 011 00000608
AC=B 00B 00021A10       AD=B 006 000029A0
AE=F FFF 80000000       AF=B 035 00000398
```

Figure 9-4. Batch Debug Session

(Continued)

(Continued)

```
X0=0000B034  00040243      X1=0000B035  00000398
X2=00000000  00000000      X3=00000000  00000064
X4=FFFFFFFF  FFFFFE0C      X5=00000000  00000000
X6=00000000  0000000F      X7=00000000  00989680
X8=00000000  00000022      X9=00000000  00000012
XA=00000000  0000004E      XB=00000000  00000000
XC=00000000  000004CC      XD=00000000  00000003
XE=00000000  00000751      XF=00000000  00000000
 CI run
 --  DEBUG: break PROC_P2, execution at M=m L=16
 CI display_breaks
 --  Break PROG_MAIN
 --  event(s) = execution
 --  location: M=module_main L=19
 --  Break PROC_P1
 --  event(s) = execution
 --  location: M=m L=14
 --  Break PROC_P2
 --  event(s) = execution
 --  location: M=m L=16
 --  Break PROC_MULT
 --  event(s) = execution
 --  location: M=perform_integer_multiplications L=7
 CI delete_breaks break=(prog_main,proc_p2)
 CI display_breaks
 --  Break PROC_P1
 --  event(s) = execution
 --  location: M=m L=14
 --  Break PROC_MULT
 --  event(s) = execution
 --  location: M=perform_integer_multiplications L=7
```

Figure 9-4. Batch Debug Session

(Continued)

(Continued)

```
 CI run
 --  DEBUG: break PROC_MULT, execution at
M=perform_integer_multiplications L=7
 CI display_calls
 --  Traceback from procedure MULT module
PERFORM_INTEGER_MULTIPLICATIONS at line 7
 --  Called from procedure P module M at line 13 byte offset 32
 --  Called from procedure MAIN module MODULE_MAIN at line 18 byte
offset 40
 CI display_calls start=2
 --  Traceback from procedure P module M at line 14
 --  Called from procedure MAIN module MODULE_MAIN at line 18 byte
offset 40
 CI run
 --  DEBUG: break PROC_P1, execution at M=m L=14
 CI display_calls
 --  Traceback from procedure P module M at line 14
 --  Called from procedure MAIN module MODULE_MAIN at line 18 byte
offset 40
 CI display_program_value name=b
b =  FALSE
 CI display_program_value name=b module=module_main procedure=main
b =  FALSE
 CI run
 --  DEBUG: program terminated by returning
 CI quit
 --  DEBUG: QUIT terminated task
 CI delete_file_connection $output session
 CI copy_file session
 EOI ENCOUNTERED.
```

Figure 9-4. Batch Debug Session

Appendixes

Glossary A

A

Access Attribute

Characteristic of a variable that determines whether the variable can be both read and written. Specifying the access attribute READ makes the variable a read-only variable.

Active Call Chain

List of calls that led to the current procedure.

Alphabetic Character

One of the following letters:

A to Z

a to z

See Character and Alphanumeric Character.

Alphanumeric Character

Alphabetic character or a digit. See Character, Alphabetic Character, and Digit.

Assignment Statement

Statement that assigns a value to a variable.

B

Batch Debugging

Debugging when the user has no direct control of debugging during program execution. Contrast with Interactive Debugging.

Bit

Binary digit. A bit has the value 0 or 1. See Byte.

Boolean

Kind of value that is evaluated as TRUE or FALSE.

Break

The primary mechanism for Debug to gain control from an executing program. A break specifies an event and an address range such that when the event occurs within the address range, Debug takes control.

Byte

Group of bits. For NOS/VE, one byte is equal to 8 bits. An ASCII character code uses the rightmost 7 bits of one byte.

Byte Offset

A number corresponding to the number of bytes beyond the beginning of a line, procedure, module, or section.

C

Character

Letter, digit, space, or symbol that is represented by a code in one or more of the standard character sets.

It is also referred to as a byte when used as a unit of measure to specify block length, record length, and so forth.

A character can be a graphic character or a control character. A graphic character is printable; a control character is nonprintable and is used to control an input or output operation.

Character Constant

Fixed value that represents a single character.

Comment

Any character or sequence of characters that is preceded by an opening brace and terminated by a closing brace or an end of line. A comment is treated exactly as a space.

Compilation Time

Time at which a source program is translated by the compiler to an object program that can be loaded and executed. Contrast with Execution Time.

Compiler

A processor that accepts source code as input and generates object code as output.

Condition Handler

Procedure called when an exception condition occurs. Condition handler processing occurs after Debug processing if Debug mode is on. The procedure is called only if it has been established as the condition handler for the condition type and the condition occurs within its scope.

D

Delimiter

Indicator that separates and organizes data.

Digit

One of the following characters:

0 1 2 3 4 5 6 7 8 9

E

Entry Point

Point in a module at which execution of the module can begin.

Event

Condition, such as division by zero, that causes Debug to gain control.

Execution Time

The time at which a compiled source program is executed. Also known as Run Time.

Expression

Notation that represents a value. A constant or variable appearing alone, or combinations of constants, variables, and operators.

External Reference

Call to an entry point in another module.

F

Field

Subdivision of a record that is referenced by name. For example, the field NORMAL in a record named OLD_STATUS is referenced as follows:

```
OLD_STATUS.NORMAL
```

I

Integer Constant

One or more digits and, for hexadecimal integer constants, the following characters:

A B C D E F a b c d e f

A hexadecimal integer constant must begin with a digit. A preceding sign and subsequent radix are optional.

Interactive Debugging

Debugging when the user has direct control of the debugging process. Contrast with Batch Debugging.

L

Load Module

Module reformatted for code sharing and efficient loading. When the user generates an object library, each object module in the module list is reformatted and written as a load module on the object library.

M

Machine Addressing

Use of actual machine addresses. Contrast with Module Addressing and Symbolic Addressing.

Machine-Level Debugging

Debugging using machine-level terms such as machine addresses. A knowledge of machine architecture is required. Contrast with Symbolic Debugging.

Module

Unit of text accepted as input by the loader, linker, or object library generator. See Object Module and Load Module.

Module Addressing

Use of addresses in terms of module and procedure names and an offset. Contrast with Machine Addressing and Symbolic Addressing.

N

Name

Combination of from 1 through 31 characters chosen from the following set.

- Alphabetic characters (A through Z and a through z).
- Digits (0 through 9).
- Special characters (#, @, $, and _).

The first character of a name cannot be a digit.

O

Object Code

Executable code produced by a compiler.

Object Module

Compiler-generated unit containing object code and instructions for loading the object code. It is accepted as input by the system loader and the object library generator.

P

Page

Allocatable unit of real memory.

Pointer

Virtual address of a value.

R

Range

Value represented as two values separated by an ellipsis. The element is associated with the values from the first value through the second value. The first value must be less than the second value. For example:

```
value..value
```

Reserved Word

Word having a predefined meaning in a language. The user cannot define a new meaning or use for a reserved word.

Ring

Level of hardware protection given a file or segment. A file is protected from unauthorized access by tasks executing in higher rings.

Run Time

See Execution Time.

S

Segment

One or more pages assigned to a file. The segment has the ring attributes of the file.

Source Code

Statements written for input to a compiler.

Statement List

One or more statements separated by delimiters.

String Constant

Sequence of characters delimited by apostrophes ('). An apostrophe can be included in the string by specifying two consecutive apostrophes.

Symbolic Addressing

Use of addresses in source program terms such as program names and line numbers. Contrast with Machine Addressing and Module Addressing.

Symbolic Debugging

Debugging using source program terms such as line numbers and program names. Contrast with Machine-Level Debugging.

T

Traceback

List of procedure names within a program, beginning with the currently executing procedure, proceeding backward through the sequence of called procedures, and ending with the main program.

V

Variable

Represents a data value.

Variable Attribute

Characteristic of a variable.

See Access Attribute.

Character Set B

This appendix lists the ASCII character set.

NOS/VE supports the American National Standards Institute (ANSI) standard ASCII character set (ANSI X3.4-1977). NOS/VE represents each 7-bit ASCII code in an 8-bit byte. The 7 bits are right-justified in each byte. For ASCII characters, the leftmost bit is always zero.

In addition to the 128 ASCII characters, NOS/VE allows use of the leftmost bit in an 8-bit byte for 256 characters. The use and interpretation of the additional 128 characters is user-defined.

Table B-1. ASCII Character Set

ASCII Code			Graphic or Mnemonic	Name or Meaning
Decimal	Hexadecimal	Octal		
000	00	000	NUL	Null
001	01	001	SOH	Start of heading
002	02	002	STX	Start of text
003	03	003	ETX	End of text
004	04	004	EOT	End of transmission
005	05	005	ENQ	Enquiry
006	06	006	ACK	Acknowledge
007	07	007	BEL	Bell
008	08	010	BS	Backspace
009	09	011	HT	Horizontal tabulation
010	0A	012	LF	Line feed
011	0B	013	VT	Vertical tabulation
012	0C	014	FF	Form feed
013	0D	015	CR	Carriage return
014	0E	016	SO	Shift out
015	0F	017	SI	Shift in
016	10	020	DLE	Data link escape
017	11	021	DC1	Device control 1
018	12	022	DC2	Device control 2
019	13	023	DC3	Device control 3
020	14	024	DC4	Device control 4
021	15	025	NAK	Negative acknowledge
022	16	026	SYN	Synchronous idle
023	17	027	ETB	End of transmission block
024	18	030	CAN	Cancel
025	19	031	EM	End of medium
026	1A	032	SUB	Substitute
027	1B	033	ESC	Escape
028	1C	034	FS	File separator
029	1D	035	GS	Group separator
030	1E	036	RS	Record separator
031	1F	037	US	Unit separator
032	20	040	SP	Space
033	21	041	!	Exclamation point
034	22	042	"	Quotation marks
035	23	043	#	Number sign
036	24	044	$	Dollar sign
037	25	045	%	Percent sign
038	26	046	&	Ampersand
039	27	047	'	Apostrophe
040	28	050	(	Opening parenthesis
041	29	051	)	Closing parenthesis
042	2A	052	*	Asterisk
043	2B	053	+	Plus
044	2C	054	,	Comma
045	2D	055	-	Hyphen
046	2E	056	.	Period
047	2F	057	/	Slant

(Continued)

Table B-1. ASCII Character Set *(Continued)*

ASCII Code				
Decimal	**Hexadecimal**	**Octal**	**Graphic or Mnemonic**	**Name or Meaning**
048	30	060	0	Zero
049	31	061	1	One
050	32	062	2	Two
051	33	063	3	Three
052	34	064	4	Four
053	35	065	5	Five
054	36	066	6	Six
055	37	067	7	Seven
056	38	070	8	Eight
057	39	071	9	Nine
058	3A	072	:	Colon
059	3B	073	;	Semicolon
060	3C	074	<	Less than
061	3D	075	=	Equals
062	3E	076	>	Greater than
063	3F	077	?	Question mark
064	40	100	@	Commercial at
065	41	101	A	Uppercase A
066	42	102	B	Uppercase B
067	43	103	C	Uppercase C
068	44	104	D	Uppercase D
069	45	105	E	Uppercase E
070	46	106	F	Uppercase F
071	47	107	G	Uppercase G
072	48	110	H	Uppercase H
073	49	111	I	Uppercase I
074	4A	112	J	Uppercase J
075	4B	113	K	Uppercase K
076	4C	114	L	Uppercase L
077	4D	115	M	Uppercase M
078	4E	116	N	Uppercase N
079	4F	117	O	Uppercase O
080	50	120	P	Uppercase P
081	51	121	Q	Uppercase Q
082	52	122	R	Uppercase R
083	53	123	S	Uppercase S
084	54	124	T	Uppercase T
085	55	125	U	Uppercase U
086	56	126	V	Uppercase V
087	57	127	W	Uppercase W
088	58	130	X	Uppercase X
089	59	131	Y	Uppercase Y
090	5A	132	Z	Uppercase Z
091	5B	133	[	Opening bracket

(Continued)

Table B-1. ASCII Character Set *(Continued)*

ASCII Code				
Decimal	Hexadecimal	Octal	Graphic or Mnemonic	Name or Meaning
092	5C	134	\	Reverse slant
093	5D	135	]	Closing bracket
094	5E	136	^	Circumflex
095	5F	137	_	Underline
096	60	140	`	Grave accent
097	61	141	a	Lowercase a
098	62	142	b	Lowercase b
099	63	143	c	Lowercase c
100	64	144	d	Lowercase d
101	65	145	e	Lowercase e
102	66	146	f	Lowercase f
103	67	147	g	Lowercase g
104	68	150	h	Lowercase h
105	69	151	i	Lowercase i
106	6A	152	j	Lowercase j
107	6B	153	k	Lowercase k
108	6C	154	l	Lowercase l
109	6D	155	m	Lowercase m
110	6E	156	n	Lowercase n
111	6F	157	o	Lowercase o
112	70	160	p	Lowercase p
113	71	161	q	Lowercase q
114	72	162	r	Lowercase r
115	73	163	s	Lowercase s
116	74	164	t	Lowercase t
117	75	165	u	Lowercase u
118	76	166	v	Lowercase v
119	77	167	w	Lowercase w
120	78	170	x	Lowercase x
121	79	171	y	Lowercase y
122	7A	172	z	Lowercase z
123	7B	173	{	Opening brace
124	7C	174	|	Vertical line
125	7D	175	}	Closing brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

Reserved Words C

The following are reserved words in CYBIL.

ALIAS	LIST	STRING
ALIGNED	LISTALL	STRLENGTH
ALLOCATE	LISTCTS	SUCC
AND	LISTEXT	THEN
ARRAY	LISTOBJ	TITLE
BEGIN	LOWERBOUND	TO
BOOLEAN	LOWERVALUE	TRUE
BOUND	MOD	TYPE
CASE	MODEND	UNTIL
CASEND	MODULE	UPPERBOUND
CAT	NEWTITLE	UPPERVALUE
CELL	NEXT	VAR
CHAR	NIL	WHILE
CHKALL	NOCOMPILE	WHILEND
CHKNIL	NOT	WRITE
CHKRNG	OF	XDCL
CHKSUB	OFF	XOR
CHKTAG	OLDTITLE	XREF
CHR	ON	#ADDRESS
COMMENT	OR	#CALLER_ID
COMPILE	ORD	#COMPARE_SWAP
CONST	PACKED	#CONVERT_POINTER_TO_PROCEDURE
CYCLE	POP	#FREE_RUNNING_CLOCK
DIV	PRED	#GATE
DO	PROCEDURE	#HASH_SVA
DOWNTO	PROCEND	#INLINE
EJECT	PROGRAM	#KEYPOINT
ELSE	PUSH	#LOC
ELSEIF	READ	#OFFSET
END	REAL	#PREVIOUS_SAVE_AREA
EXIT	RECEND	#PTR
FALSE	RECORD	#PURGE_BUFFER
FOR	REL	#READ_REGISTER
FOREND	REP	#REL
FREE	REPEAT	#RING
FUNCEND	RESET	#SCAN
FUNCTION	RETURN	#SEGMENT
HEAP	RIGHT	#SIZE
IF	SECTION	#TRANSLATE
IFEND	SEQ	#UNCHECKED_CONVERSION
IN	SET	#WRITE_REGISTER
INLINE	SKIP	$CHAR
INTEGER	SPACING	$INTEGER
LEFT	STATIC	$REAL

Data Representation in Memory D

Memory is made up of 8-bit addressable bytes with eight bytes to one 64-bit word. (An 8-bit byte is synonymous with a cell.) Table D-1 summarizes how different data types are represented in memory. The alignment column indicates how a variable of the data type is stored in packed and unpacked format. The word "byte" means it is stored in the first available byte; "bit" means it is stored in the first available bit.

Table D-1. Data Representation in Memory

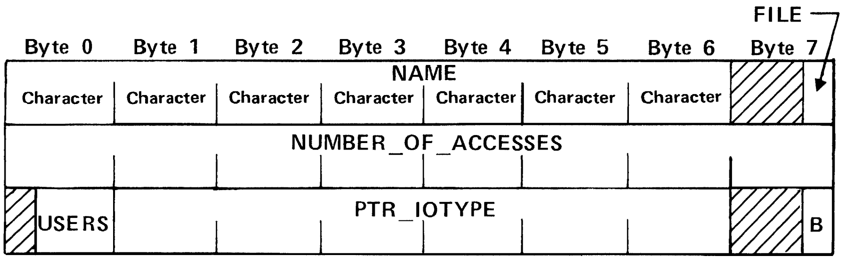
		Alignment	
Type	**Size**	**Unpacked**	**Packed**
Integer	8 bytes	Byte	Byte
Character	1 byte	Byte	Bit
Boolean	1 bit	Right-justified in a byte	Bit
Ordinal	As needed	Right-justified for components	Bit in a byte
Subrange	As needed for components	Right-justified in a byte	Bit
Real	8 bytes	Byte	Byte
Cell	Byte	Byte	Byte
Fixed pointer	6 bytes	Byte	Byte
Fixed relative pointer	4 bytes	Byte	Byte
String	1 byte for each character	Byte	Byte
Array	Depends on type of components	Byte	Components are unaligned
Set	As needed for components	Right-justified in a byte	Bit if < 57 components; byte if > 57 components

The following examples show how a record would look in memory in various formats: first unpacked, then packed, packed with some positioning changes, and finally aligned. The memory shown here is in eight-byte words, but because bytes can be addressed individually, it's possible the record could start at any byte (if it is not aligned otherwise).

The unpacked record is:

```
TYPE
  table = RECORD
            name : string(7),
            file : (bi, di, lg, pr),
            number_of_accesses : integer,
            users : 0..100,
            ptr_iotype : ^iotype,
            b : boolean,
          RECEND;
```

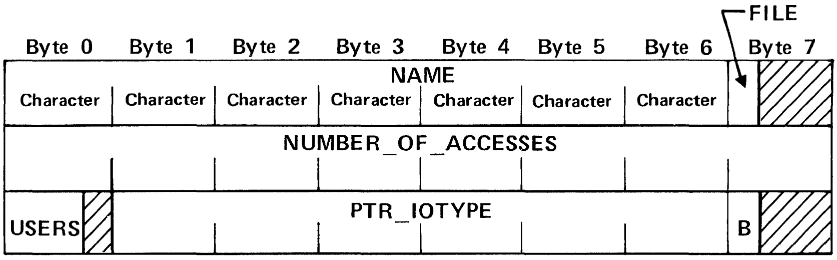
This record would appear in memory as follows (slashes indicate unused memory):



The packed record is:

```
TYPE
  table = PACKED RECORD
            name : string(7),
            file : (bi, di, lg, pr),
            number_of_accesses : integer,
            users : 0..100,
            ptr_iotype : ^iotype,
            b : boolean,
          RECEND;
```

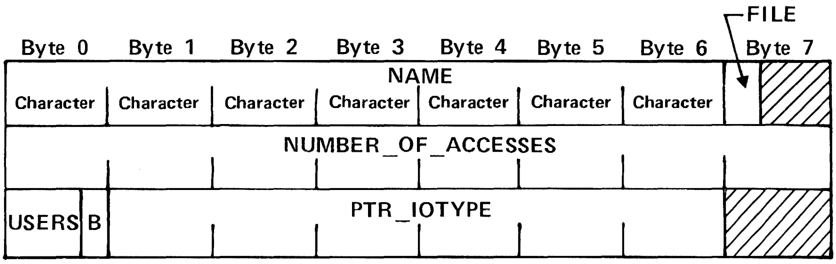
This record would appear in memory as follows (slashes indicate unused memory):



The record, as follows, is now rearranged slightly to make more efficient use of the space.

```
TYPE
  table = PACKED RECORD
                   name : string(7),
                   file : (bi, di, lg, pr),
                   number_of_accesses : integer,
                   users : 0..100,
                   b : boolean,
                   ptr_iotype : ^iotype,
                 RECEND;
```

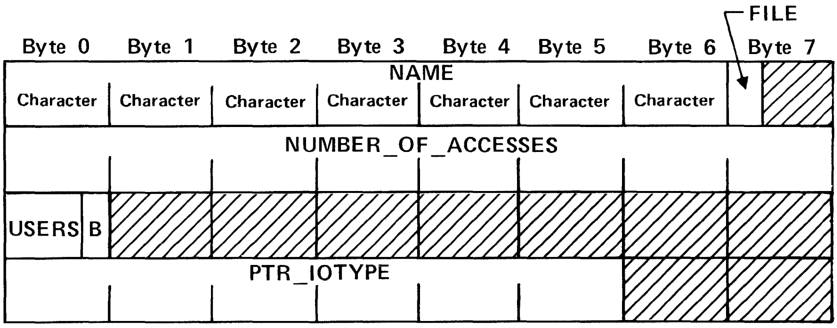
This record would appear in memory as follows (slashes indicate unused memory):



The following record declares the pointer field to be aligned at byte zero (the first byte) of a word.

```
TYPE
  table = PACKED RECORD
                    name : string(7),
                    file : (bi, di, lg, pr),
                    number_of_accesses : integer,
                    users : 0..100,
                    b : boolean,
                    ptr_iotype : ALIGNED [0 MOD 8] ^iotype,
                  RECEND;
```

This record would appear in memory as follows (slashes indicate unused memory):



Index

Index

CYBIL for NOS/VE Language Definition 60464113 B

We would like your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

Who Are You?	**How Do You Use This Manual?**	**Which Do You Also Have?**
☐ Manager	☐ As an Overview	☐ Any SCL Manuals
☐ Systems Analyst or Programmer	☐ To Learn the Product/System	☐ CYBIL File Interface
☐ Applications Programmer	☐ For Comprehensive Reference	☐ CYBIL System Interface
☐ Operator	☐ For Quick Look-up	
☐ Other ______		

What programming languages do you use? ______

Which are helpful to you? ☐ Procedures Index (inside covers) ☐ Glossary ☐ Related Manuals page

☐ Character Set ☐ Other: ______

How Do You Like This Manual? Check those that apply.

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple ☐ Too complex)
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you what you need to know about the topic?

Comments? If applicable, note page number and paragraph.

Would you like a reply? ☐ Yes ☐ No

Continue on other side

From:

Name ______ Company ______

Address ______ Date ______

______ Phone No. ______

______

Please send program listing and output if applicable to your comment.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
ARH219
4201 North Lexington Avenue
Saint Paul, Minnesota 55112

FOLD FOLD

Comments (continued from other side)

(Continued from inside front cover)

Procedure Index

Compilation Index

Debug Command and Function Index